

EXPERT INSIGHT

# Mastering Microsoft Dynamics 365 Business Central

The complete guide for designing and integrating  
advanced Business Central solutions

**Second Edition**



**Stefano Demiliani**  
**Duilio Tacconi**

**<packt>**

# **Mastering Microsoft Dynamics 365 Business Central**

Second Edition

The complete guide for designing and integrating advanced  
Business Central solutions

**Stefano Demiliani**

**Duilio Tacconi**



BIRMINGHAM—MUMBAI

# Mastering Microsoft Dynamics 365 Business Central

Second Edition

Copyright © 2024 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Senior Publishing Product Manager:** Larissa Pinto

**Acquisition Editors – Peer Reviews:** Gaurav Gavas and Jane Dsouza

**Project Editor:** Parvathy Nair

**Content Development Editor:** Matthew Davies

**Copy Editor:** Safis Editing

**Technical Editor:** Kushal Sharma

**Proofreader:** Safis Editing

**Indexer:** Hemangini Bari

**Presentation Designer:** Ganesh Bhadwalkar

**Developer Relations Marketing Executive:** Sohini Ghosh

First published: December 2019

Second edition: March 2024

Production reference: 1130324

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83763-064-6

[www.packt.com](http://www.packt.com)

*I dedicate this book to my family and especially to my little daughter, Sara. Love you,  
thanks for the energy you give me every day...*

*– Stefano Demiliani*

*This book is dedicated to the love of my life: Laura.*

*– Duilio Tacconi*

# Contributors

## About the authors

**Stefano Demiliani** is a Microsoft MVP for Business Applications and Azure, MCT Regional Lead and a long-time expert on Microsoft technologies. He works as a CTO for EID NAV-lab Group and his main activities are architecting solutions with Azure and Dynamics 365 ERPs. He's the author of many IT books for Packt and a frequent speaker on international conferences about Azure, Dynamics 365, and Power Platform. You can reach him on Twitter or on LinkedIn or via his personal website.

**Duilio Tacconi** is a performance specialist for EOS Solutions Group who has worked with ERP technologies since 1998. As a Microsoft MCT and author of educational resources about Microsoft, Duilio has been passionate about Microsoft Dynamics NAV and 365 Business Central since 2004. In the past, Duilio worked as a former Microsoft senior escalation engineer for 15 years, from 2008 to 2023.

Duilio has handled thousands of support requests resolving issues with Microsoft products over the years. Currently, he implements processes to analyze performance data through telemetries and insights tools, as well as being involved in the continuous updating of clients from on-premises to online.

## About the reviewers

**Daniel Rimmelzwaan** has, over the past 25 years, worked in almost every role implementing Business Central. He currently owns his own business (risplus.com) performing business analysis, architecture and design, and development. This is the tenth book Daniel has reviewed for Packt.

Ever since he started working with Business Central, Daniel has contributed to its online communities. For those contributions, Daniel has received eighteen Microsoft MVP awards between 2005 and 2022. You can follow Daniel's blog at <https://thedenster.com>.

**Danilo Capuano** is a technical delivery manager and office manager at Agic Technology. He is a **Microsoft Certified Trainer (MCT)** and MVP. Danilo specializes in solution architecture for Power Platform, Dynamics 365, and Azure, as well as being familiar with DevOps engineering. He is active in the Microsoft community in Italy, being a group leader for Power Apps User Group Italia and Power Pages User Group Italia. He is active on social media and can be found through his blog ([danilocapuano.blog](http://danilocapuano.blog)), Twitter (@capuanodanilo), and LinkedIn (/capuanodanilo). He has previously worked as a technical reviewer on several other titles, including *Fundamentals of CRM with Dynamics 365 and Power Platform*, *Mastering Microsoft Dynamics NAV 2016*, *Microsoft Dynamics NAV 7 Programming Cookbook*, *Microsoft Dynamics NAV 2013 Application Design*, *Microsoft Power Platform Enterprise Architecture*, *Learn Microsoft PowerApps*, and *Programming Microsoft Dynamics NAV 2015*.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>



# Table of Contents

<b>Preface</b>	<b>xvii</b>
<b>Chapter 1: Microsoft Dynamics 365 Business Central's Online Momentum</b>	<b>1</b>
Dynamics 365 Business Central's evolution .....	3
The Universal Code initiative .....	10
The role of open source and social networks .....	12
The future perspective .....	16
Summary .....	19
<b>Chapter 2: Mastering a Modern Development Environment</b>	<b>21</b>
The Visual Studio Code user interface .....	22
Code editor • 23	
Status bar • 24	
Activity bar • 25	
<i>Manage • 25</i>	
<i>Command Palette • 25</i>	
Sidebar • 26	
<i>EXPLORER (Ctrl + Shift + E) • 26</i>	
<i>SEARCH (Ctrl + Shift + F) • 28</i>	
<i>SOURCE CONTROL (Ctrl + Shift + G) • 28</i>	
<i>DEBUG (Ctrl + Shift + D) • 29</i>	
<i>EXTENSIONS (Ctrl + Shift + X) • 29</i>	
Panels area • 30	
<i>PROBLEMS • 31</i>	
<i>OUTPUT • 31</i>	
<i>DEBUG CONSOLE • 32</i>	
<i>TERMINAL • 32</i>	

<b>Visual Studio Code – the editing features .....</b>	<b>32</b>
Comment lines • 33	
Delimiter matching • 33	
Text selection • 33	
Code block folding • 34	
Multiple cursors (or multi-cursors) • 34	
Mini-map • 35	
Breadcrumbs • 35	
IntelliSense • 36	
Word completion • 36	
Go to definition • 36	
Find all references • 37	
Peek definition • 37	
Renaming symbols • 38	
<b>Understanding the AL Language extension .....</b>	<b>39</b>
AL Language • 39	
<i>launch.json</i> • 42	
<i>app.json</i> • 45	
<i>Understanding symbols</i> • 51	
<i>Inside symbols</i> • 52	
AL Language extension settings • 56	
<b>Understanding code analyzers .....</b>	<b>58</b>
<b>GitHub Copilot for AL developers .....</b>	<b>61</b>
<b>Summary .....</b>	<b>63</b>
 <b>Chapter 3: Extension Development Fundamentals</b>	 <b>65</b>
<b>Technical requirements .....</b>	<b>65</b>
<b>Basic concepts regarding extensions .....</b>	<b>66</b>
<b>Understanding the basics of AL .....</b>	<b>69</b>
Creating a new workspace • 70	
Defining objects using snippets • 70	
Table object definition • 74	
Page object definition • 77	
Table extension object definition • 80	
Page extension object definition • 82	
Codeunit object definition • 84	

Event object definitions • 85	
XMLport object definition • 89	
Query object definition • 91	
Enum object definition • 94	
Profile object definition • 97	
<b>Understanding AL project structure best practices .....</b>	<b>98</b>
<b>Naming guidelines and AL object ranges .....</b>	<b>101</b>
<b>Working on AL coding guidelines .....</b>	<b>103</b>
<b>Summary .....</b>	<b>106</b>
 <b>Chapter 4: Developing a Customized Solution for Dynamics 365 Business Central</b>	 <b>109</b>
<b>Translating a business case into a real-world extension .....</b>	<b>109</b>
Developing the Dynamics 365 Business Central customization • 111	
Customer category implementations • 114	
<i>Table definition • 114</i>	
<i>Page definition • 118</i>	
<i>The tableextension definition • 124</i>	
<i>The pageextension definition • 126</i>	
<i>Codeunit definition • 129</i>	
<i>Handling event subscribers • 132</i>	
Gift campaign implementations • 133	
<i>Table definition • 133</i>	
<i>Page definition • 135</i>	
<i>Codeunit definition • 139</i>	
Vendor quality implementations • 145	
<i>Table definition • 145</i>	
<i>Page definition • 148</i>	
<i>The pageextension definition • 151</i>	
<i>Codeunit definition • 154</i>	
<b>Promoting actions .....</b>	<b>158</b>
<b>Creating page views .....</b>	<b>160</b>
<b>Installing and upgrading codeunits .....</b>	<b>162</b>
<b>Defining permission sets in AL .....</b>	<b>169</b>
<b>Summary .....</b>	<b>172</b>

<b>Chapter 5: Writing Code for Extensibility</b>	<b>173</b>
Why do we need extensible code? .....	173
Business scenario .....	175
Events and the “Handled” pattern .....	175
Writing a dependent extension .....	183
Interfaces in AL .....	188
Extension’s code protection .....	193
Summary .....	194
<b>Chapter 6: Advanced AL Development</b>	<b>197</b>
Understanding immutable keys .....	197
Access modifiers in AL .....	198
Handling errors with TryFunctions .....	202
Using collectible errors .....	203
Handling actions on errors .....	206
Creating and extending role centers .....	207
Customizing the headline • 211	
Handling XML and JSON files with the AL language .....	214
Understanding Isolated Storage .....	218
Working with control add-ins .....	222
Creating a PDF-Viewer control add-in • 223	
Notifications inside Dynamics 365 Business Central .....	227
Understanding page background tasks .....	230
Using Azure Key Vault in AL extensions .....	236
Namespaces in AL language .....	239
Summary .....	242
<b>Chapter 7: Handling Files with Dynamics 365 Business Central</b>	<b>243</b>
Handling files with AL .....	243
Handling attachments • 249	
Reading and writing text data from blob fields • 251	
Using the Media and MediaSet data types in AL code • 252	
Using XMLport in AL code • 254	
Handling XML and JSON files with AL • 256	
Using persistent blobs .....	260
Isolated Storage .....	262

<b>Using Azure Blob Storage from AL .....</b>	<b>265</b>
Creating a storage account in Azure • 267	
Using Azure Blob Storage from AL code • 268	
Handling Dynamics 365 Business Central attachments in Azure Blob Storage • 270	
<b>Using Azure file shares from AL .....</b>	<b>275</b>
<b>Summary .....</b>	<b>278</b>

---

## **Chapter 8: Report Development 281**

---

<b>Anatomy of the report object .....</b>	<b>281</b>
<b>The report extension object .....</b>	<b>284</b>
<b>Tools to use for RDL, Word, and Excel layouts .....</b>	<b>284</b>
The RDL, Word, and Excel layout features • 285	
Part 1 – Designing the dataset • 285	
Part 2 – Creating a simple RDL layout • 290	
<i>Part 2.1 – Creating the RDL report header • 290</i>	
<i>Part 2.2 – Adding a table control to the RDL report body • 293</i>	
Part 3 – Understanding grouping • 296	
Part 4 – Building a simple request page • 299	
Part 5 – Adding database images • 300	
Part 6 – Adding a Word layout • 303	
Part 7 – Adding an Excel layout • 308	
<b>Report extension object: a basic example .....</b>	<b>313</b>
<b>Cloning and refactoring reports .....</b>	<b>317</b>
<b>Feature limitations when developing RDL or Word layout document reports .....</b>	<b>322</b>
<b>Understanding report performance considerations .....</b>	<b>323</b>
<b>Summary .....</b>	<b>324</b>

---

## **Chapter 9: Printing 327**

---

<b>Understand cloud-ready printing .....</b>	<b>328</b>
<b>Email printers .....</b>	<b>329</b>
<b>Microsoft Universal Print printers .....</b>	<b>330</b>
Universal Print connector • 330	
Universal Print portal • 332	
Universal Print Integration extension • 335	
<b>Deep dive into the modern printing structure .....</b>	<b>338</b>
<b>Alternatives to Microsoft Universal Print .....</b>	<b>345</b>
<b>Summary .....</b>	<b>346</b>

<b>Chapter 10: Debugging</b>	<b>347</b>
Running in debug mode .....	347
<i>Debugging logs: verbose mode</i> • 350	
Visual Studio Code debugger sections .....	351
Debugger sidebar • 352	
<i>VARIABLES</i> • 352	
<i>Watch</i> • 354	
<i>Callstack</i> • 354	
<i>Breakpoints</i> • 354	
Debugger toolbar • 354	
Debugging in attach mode • 355	
Non-debuggable items • 356	
Snapshot debugging .....	358
Performance profiling .....	363
Summary .....	375
<b>Chapter 11: Telemetry</b>	<b>377</b>
Signal fundamentals .....	377
Dimensions • 378	
Aggregated signals • 379	
Microsoft's recommendations • 380	
Example • 381	
Enabling partner telemetry in Dynamics 365 Business Central online .....	381
KQL log analysis .....	384
Statements • 386	
Operators • 387	
Functions • 387	
Application Insights .....	394
Alerts • 394	
Dashboards • 397	
Workbooks • 402	
Tools to analyze telemetry data .....	406
Power BI telemetry apps .....	409
Custom signals .....	414
Summary .....	418

<b>Chapter 12: Coding for Performance</b>	<b>419</b>
Defining an efficient data access layer .....	419
Table extension changes from Dynamics 365 Business Central version 23 • 423	
Setting the transaction isolation level in AL code • 424	
Writing efficient pages and reports • 427	
Writing performant installation and upgrade AL codeunits • 428	
Events and performance .....	430
Running asynchronous patterns .....	430
Using StartSession in AL code • 430	
Using Task Scheduler in AL code • 432	
Testing and validating performances .....	433
Summary .....	436
<b>Chapter 13: Dynamics 365 Business Central APIs</b>	<b>437</b>
Using the OData protocol for APIs .....	438
Configuring OAuth authentication for Dynamics 365 Business Central APIs .....	439
Registering an application in Microsoft Entra ID • 440	
Setting the application permissions • 442	
Creating a client secret • 444	
Microsoft Entra application registration in Dynamics 365 Business Central • 445	
Acquiring an authentication token from Microsoft Entra ID .....	448
Using Dynamics 365 Business Central standard APIs .....	450
Creating a custom API in Dynamics 365 Business Central .....	459
Implementing a new API for a custom entity • 460	
Implementing a new API for an existing entity • 467	
Using the read-only database replica • 471	
Dynamics 365 APIs' operational limits • 471	
Using OData bound actions .....	472
Using OData unbound actions .....	474
Using OData batch calls with Dynamics 365 Business Central APIs • 476	
Using Dynamics 365 Business Central webhooks .....	482
Summary .....	485

<b>Chapter 14: Extending Dynamics 365 Business Central with Azure Services</b>	<b>487</b>
Overview of Azure Functions .....	487
Creating functions with Azure Functions .....	489
Using Azure Functions from AL .....	501
Overview of Azure Logic Apps .....	504
Creating workflows with Azure Logic Apps .....	506
Summary .....	510
<b>Chapter 15: DevOps for Dynamics 365 Business Central</b>	<b>511</b>
AL-Go for GitHub: an introduction .....	511
Creating a new per-tenant extension with AL-Go for GitHub .....	514
Branching strategies .....	517
Master-only branch • 518	
Feature/developer branches • 518	
Release branching • 519	
Other strategies • 519	
Git flow • 519	
GitHub flow • 520	
Git merge strategies .....	521
Fast-forward merge • 521	
Squash commit • 521	
Rebase • 522	
Git in Visual Studio Code .....	523
Visual Studio Code GUI for Git • 524	
Workflow • 525	
Handling the CI/CD pipeline .....	526
Setting up your self-hosted GitHub runner .....	529
Handling dependencies between applications .....	533
Adding a test application to an existing project .....	538
Registering a customer sandbox environment for continuous deployment .....	539
Creating a release for your application .....	543
Registering a customer Production environment for manual deployment .....	544
Adding a performance test app to your repository .....	546
Using AL-Go for GitHub for AppSource development .....	547
Summary .....	551

<b>Chapter 16: Dynamics 365 Business Central and Power Platform Integration</b>	<b>553</b>
Technical requirements .....	553
Power Automate and Dynamics 365 Business Central .....	553
Example 1: Adding a default image to a Customer record • 554	
Example 2: Exporting a selected invoice as PDF to OneDrive • 559	
Creating a Power Apps app with Dynamics 365 Business Central integration .....	562
Scenario: Expense app with offline capability • 563	
Part 1: Creation of the canvas app project • 565	
Part 2: Establishing a data connection to Dynamics 365 Business Central • 566	
Part 3: Working with Dynamics 365 Business Central data from the canvas app • 568	
Part 4: Adding controls and business logic to the canvas app • 569	
Part 5: Adding offline capabilities to the application • 576	
Part 6: Calling a Power Automate flow to execute actions in the app • 591	
Exposing Dynamics 365 Business Central data to Dataverse by using virtual tables .....	595
Exposing Dynamics 365 Business Central events to Dataverse .....	603
Summary .....	607
<b>Chapter 17: Useful and Proficient Tools for AL Developers</b>	<b>609</b>
Who is Waldo? .....	609
What tools to use .....	610
The AL Extension Pack • 611	
The waldo's CRS AL Language Extension • 612	
Run objects • 613	
Renaming / Reorganizing files • 613	
Search on Google/Microsoft Docs • 615	
Snippets • 615	
Feedback to Waldo • 616	
Waldo GitHub repo • 616	
Free Visual Studio Code extensions for AL developers • 617	
AZ AL Dev Tools/AL Code Outline (by Andrzej Zwierzchowski) • 617	
AL Code Actions (by David Feldhoff) • 618	
AL Object ID Ninja (by Vjeko.com) • 619	
BusinessCentral.LinterCop (by Stefan Maron) • 619	
AL Toolbox (by Bart Permentier) • 619	
AL Navigator (by Waldemar Brakowski) • 619	
Free Visual Studio Code extensions for AL Language side features • 619	
Summary .....	620

---

<b>Chapter 18: Creating Generative AI Solutions for Dynamics 365 Business Central</b>	<b>621</b>
<hr/>	
Introduction to generative AI main concepts .....	622
Introducing Azure OpenAI Service .....	623
Dynamics 365 Business Central and Azure OpenAI Service .....	624
Deploy an AI model with Azure OpenAI Service .....	626
Creating a generative AI solution for Dynamics 365 Business Central .....	631
Summary .....	640
 <b>Other Books You May Enjoy</b>	 <b>645</b>
<hr/>	
<b>Index</b>	<b>649</b>
<hr/>	

# Preface

This book is an essential guide for developers that need to create advanced solutions with Dynamics 365 Business Central, the cloud-based **Enterprise Resource Planning (ERP)** solution provided by Microsoft.

We start by explaining the Dynamics 365 Business Central platform and its most recent developments, as well as anticipating its future development and how you can prepare for it. Our aim in this edition is to start coding an extension as soon as possible, so we will promptly move on to cover developer-oriented topics when coding your own Business Central extensions, covering the best coding practices for performant code.

In this edition, you'll find new and improved coverage of some of the most common developer tasks in Business Central. In particular, you'll find chapters on reporting, printing, and handling files from the cloud platform. After covering these tasks, you'll learn how to perform debugging and telemetry for fully monitoring a Dynamics 365 Business Central tenant. You'll also learn how to apply DevOps principles to your development practice by using AL-Go for GitHub.

In this book we put a strong focus on integration, covering topics like APIs, Azure Functions, Azure Logic Apps, Dataverse, and Power Platform. Integrating with these technologies allows you to create custom applications connected to Dynamics 365 Business Central. In this edition we wanted to expand our coverage of Power Apps in particular, so we have added a detailed project with step-by-step instructions. You'll learn how to create a fully functional app, complete with offline capabilities.

An exciting new topic in Microsoft is the integration of custom generative AI solutions in Dynamics 365 Business Central by using Azure OpenAI and the Copilot developer toolkit from AL. We'll wrap up covering how you can configure the Copilot capabilities in Business Central, as well as how to create your own copilots.

Overall, we hope this book will help experienced developers learn the latest tools in the trade and best coding practices to become an expert at developing solutions for Dynamics 365 Business Central.

## Who this book is for

This book is designed to help experienced professionals enhance their knowledge and understanding of Dynamics 365 Business Central. Whether you want to develop more performant extensions, learn how to handle printing solutions, or improve your use of telemetries, this book is for you.

## What this book covers

*Chapter 1, Microsoft Dynamics 365 Business Central's Online Momentum*, explains Dynamics 365 Business Central and its importance in the Microsoft's ERP ecosystem.

*Chapter 2, Mastering a Modern Development Environment*, explains everything you need to know to create a development environment for Dynamics 365 Business Central.

*Chapter 3, Extension Development Fundamentals*, explains the basics on how to create a customized solution for Dynamics 365 Business Central.

*Chapter 4, Developing a Customized Solution for Dynamics 365 Business Central*, guides you on creating a complete customized solution for Dynamics 365 Business Central using guidelines for publishing it on the Microsoft marketplace.

*Chapter 5, Writing Code for extensibility*, explains tips and best practices for creating solutions that can be extended.

*Chapter 6, Advanced AL Development*, explains how you can use advanced coding practices to enhance your customized solutions for Dynamics 365 Business Central.

*Chapter 7, Handling Files with Dynamics 365 Business Central*, explains how you can use files in a cloud environment and how you can use Azure Storage in your solutions.

*Chapter 8, Report Development*, explains how you can create reports in Dynamics 365 Business Central and how you can extend existing reports.

*Chapter 9, Printing*, explains how you can handle printer access and printer jobs from Dynamics 365 Business Central in a fully cloud-compatible way.

*Chapter 10, Debugging*, explains how you can use debugger and snapshot debugging features in Dynamics 365 Business Central to troubleshoot problems.

*Chapter 11, Telemetry*, explains how you can fully monitor a Dynamics 365 Business Central tenant by using Azure Applications Insights and the ingested telemetry data.

*Chapter 12, Coding for Performance*, gives you tips and tricks on how you can create efficient AL code with performance optimization in mind.

*Chapter 13, Dynamics 365 Business Central APIs*, explains how you can use existing APIs and how you can create custom APIs with Dynamics 365 Business Central, with also advanced usage techniques.

*Chapter 14, Extending Dynamics 365 Business Central with Azure services*, shows you how you can use Azure Functions and Azure Logic apps to extend your solutions on using cloud resources or on executing custom .NET code.

*Chapter 15, Applying DevOps in AL Extension Development*, explains the importance of applying DevOps practices in a Dynamics 365 Business Central project and shows you how to use AL-Go for GitHub for handling your development process.

*Chapter 16, Dynamics 365 Business Central and Power Platform Integration*, shows you how you can use Power Automate, Power Apps and Dataverse to create advanced solutions that integrates Dynamics 365 Business Central with the Power Platform.

*Chapter 17, Useful and Proficient Tools for AL Developers*, gives you an overview of tools that you can use to improve your development process and to increase efficiency.

*Chapter 18, Creating Generative AI Solutions for Dynamics 365 Business Central*, explains how you can integrate generative AI features into your Dynamics 365 Business Central solutions by using Azure OpenAI and the Copilot Toolkit.

## To get the most out of this book

To follow along with the topics and projects in this book, you will need to set up Dynamics 365 Business Central and Visual Studio Code:

- **Dynamics 365 Business Central:** This software is available as a SaaS service, so a registration on <https://businesscentral.dynamics.com> is required.
- **Visual Studio Code:** You can download Visual Studio Code for free from <https://code.visualstudio.com/> and install it from the VS Code marketplace. Then, additionally install the AL Language extension.

Note that we will explain all these steps in more detail later in the book.

## Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781837630646>.

## Conventions used

There are a number of text conventions used throughout this book.

**CodeInText:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “A `permissionset` object can be assignable to a user or not.”

A block of code is set as follows:

```
"features": [  
    "TranslationFile"  
]
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
begin  
    dt.SetTables(Database::FromTable, Database::ToTable);  
    dt.AddFieldValue(from.FieldNo("SmallCodeField"), to.FieldNo("SmallCodeField"));  
    dt.AddFieldValue(from.FieldNo("IntField"), to.FieldNo("IntField"));  
    dt.AddJoinCondition(from.FieldNo("id"), to.FieldNo("id"));  
    dt.CopyFields();  
end;
```

Any command-line input or output is written as follows:

```
setBreakpoints  
02/20/2023 12:12:14 [/9] Parsing Report 50111 "Item Ledger Entry Analysis".  
02/20/2023 12:12:14 [/9] Process:
```

**Bold:** Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Select **System info** from the **Administration** panel.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** Email [feedback@packtpub.com](mailto:feedback@packtpub.com) and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at [questions@packtpub.com](mailto:questions@packtpub.com).

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

## Share your thoughts

Once you've read *Mastering Microsoft Dynamics 365 Business Central, Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781837630646>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

# 1

## Microsoft Dynamics 365 Business Central's Online Momentum

Microsoft Dynamics 365 Business Central is one of the best-in-class pieces of cloud-based ERP application software that is targeted at the small and medium business market. The online application is based on the **Software-as-a-Service (SaaS)** model, and it is typically sold through **Cloud Solution Provider (CSP)** partners.

In 2023 and 2024, Dynamics 365 Business Central online has been the fastest-growing cloud ERP in the SMB segment worldwide and also nominated as overall **Best Manufacturing ERP by Forbes**: <https://www.forbes.com/advisor/business/software/best-manufacturing-erp/>

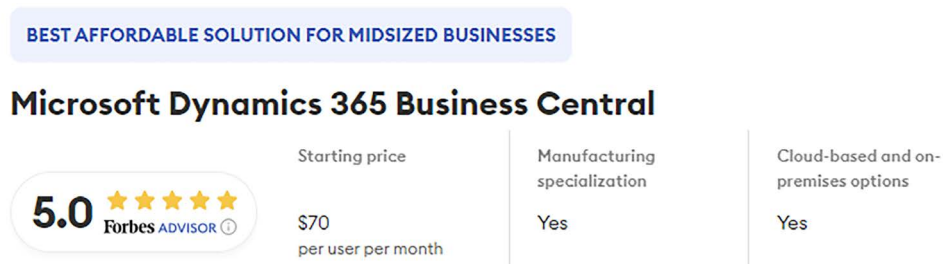


Figure 1.1: Business Central popularity with Forbes

Potential customers can subscribe for fast setup with a trial version through the following link and by providing an email address that is bound to a Microsoft 365 subscription and a phone number: <https://go.microsoft.com/fwlink/?LinkId=2143349&clid=0x409>



## You've selected Dynamics 365 Business Central

- 1 Let's get you started
 

Enter your work or school email address, we'll check if you need to create a new account for Dynamics 365 Business Central.

Email

This is required

By proceeding you acknowledge that if you use your organization's email, your organization may have rights to access and manage your data and account.

[Learn More](#)

[Next](#)
- 2 Create your account
- 3 Confirmation details



### Connected business management

- Connect sales, service, finance, and operations in a single solution
- Edit and analyze your data in Excel
- Manage sales in Outlook and share data via Teams
- Extend your solution with industry-specific apps via AppSource

Figure 1.2: Business Central fast setup

When the trial period ends, the product needs to be purchased.

Official licensing is assigned typically through Microsoft Partners that are accredited and certified by the CSP program. Simply browse <https://dynamics.microsoft.com/en-us/business-central/overview/> and click on **Find a partner** under the **Partners** tab in the breadcrumb on top of the web page, in the **Dynamics 365** section.

All of this looks as simple as 1, 2, 3, and for end-users, it definitely is.

Behind that, there is an entire world and ecosystem of interconnected people from Microsoft and partners that work on this product year after year with impressive results, making its momentum for the future so bright.

In this chapter, we will cover the following topics:

- **Dynamics 365 Business Central's evolution:** How the platform and application evolved over the years.
- **The Universal code initiative:** To write not only good code but code that can be deployed in all topologies.
- **The role of social networks and open source:** How to stay up to date with the latest news from Microsoft and from the field.
- **The future perspective:** What to potentially expect in the upcoming years and which skills are and will be needed to evolve and master Dynamics 365 Business Central.

By the end of this chapter, you'll have a clear and in-depth overview of the Microsoft Dynamics 365 Business Central platform and its ecosystem.

## Dynamics 365 Business Central's evolution

The application's core code and business processes come from the evolution of over thirty years of feature development of Microsoft Dynamics NAV (formerly known as *Navision*): one of the most solid pieces of on-premises ERP software in the SMB domain.

As an example of this evolution, let's look at how Dynamics 365 Business Central has become more internationally available over time.

If you get the chance to read the first edition of this book, at that time of writing, in 2018, Dynamics 365 Business Central was officially localized by Microsoft in just 18 countries. Starting from the October 2018 update, CSP partners can create their own localized versions for countries where Dynamics 365 Business Central has not been released officially or is not on the radar as a Microsoft localization.

These localizations start with the worldwide standard application base (so-called **W1**) and are distributed as extensions through the Microsoft Dynamics 365 Marketplace, also known as **AppSource**. Like any extension (or app) that is deployed through AppSource, all the application and technical support is provided by the partner who sells the app through the marketplace.



You can read more about this at <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/readiness/readiness-develop-localization#service-availability-in-additional-countries>.

Since October 2018, a lot of time has passed by and a lot more localizations have been provided by Microsoft and its partners. At the time we are writing, Dynamics 365 Business Central online provides:

- 24 localized versions by Microsoft
- 133 localized versions by partners
- 54 language apps

And the number is growing, release after release. Compare these numbers with just the 18 countries available in 2018 and you will have an idea of how fast Dynamics 365 Business Central is evolving.



You can read more about this at <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/readiness/readiness-develop-localization#service-availability-in-additional-countries> and <https://appsource.microsoft.com/en-us/marketplace/apps?page=1&product=dynamics-365%3Bdynamics-365-business-central&search=language>.

Considering the backend and frontend, the core design of Dynamics 365 Business Central as a mature web application relies on Microsoft Azure, Microsoft Dynamics, and Microsoft 365 platforms and their service offerings. The evolution of this ensemble is strongly and tightly connected.

Let's rewind a bit of the record and redesign here the full history of Dynamics 365 Business Central and its glorious growth.

The name *Dynamics 365 Business Central* appeared (and so it remains) for the first time with the April 2018 release version 13.x, after being called different names and having different logos. See below how the logo and name have evolved over the years:





Dynamics 365 for Financials	Dynamics 365 for Financials and Operations: Business Edition	Dynamics 365 Business Central	Dynamics 365 Business Central
Version 10	Version 12	Version 13 and 14	Version 15.x and onward
04.2017 – 11.2017	11.2017 – 10.2018	10.2018 – 10.2019	10.2019 onward
			

Table 1.1: Business Central version history

The first Dynamics 365 Business Central version was the October 2018 release 13.x, which together with version 14.x, still shared a hybrid development (with both AL and C/AL) and deployment (old legacy Windows client and modern web client) for on-premises while SaaS extension deployments could only be performed through AL-only extensions targeted for the web client.

In October 2019, with version 15.x, Microsoft made a hard cut with the past by announcing the removal of both the CSIDE Development environment and legacy Windows client, to make development and client deployment more uniform between SaaS and on-premises versions. This version is very important in Dynamics 365 Business Central's history since it is the first version bound to a *modern lifecycle policy* for both on-premises and online versions. A modern lifecycle policy involves the following:

- Major releases of the product two times per year, typically in April (called Wave 1) and in October (called Wave 2)
- One minor update every month
- Common modern lifecycle policy unified for both online and on-premises deployments.

This last bullet point is crucial since it determines a permanence in SaaS for a major version, typically up to five minor monthly updates (e.g., from 22.0 to 22.5), and for on-premises, a very short lifecycle of typically 1.5 years with just eighteen updates (e.g., from 22.0 to 22.18) before going completely out of support.

During the first five minor updates, Microsoft is also releasing the previously announced and previewed new platform and/or application features or their enhancements. After the fifth minor update, where the cumulative updates are released only for the on-premises audience, these will just contain minor bug fixes and regulatory features.

See more at <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/terms/lifecycle-policy-on-premises>.

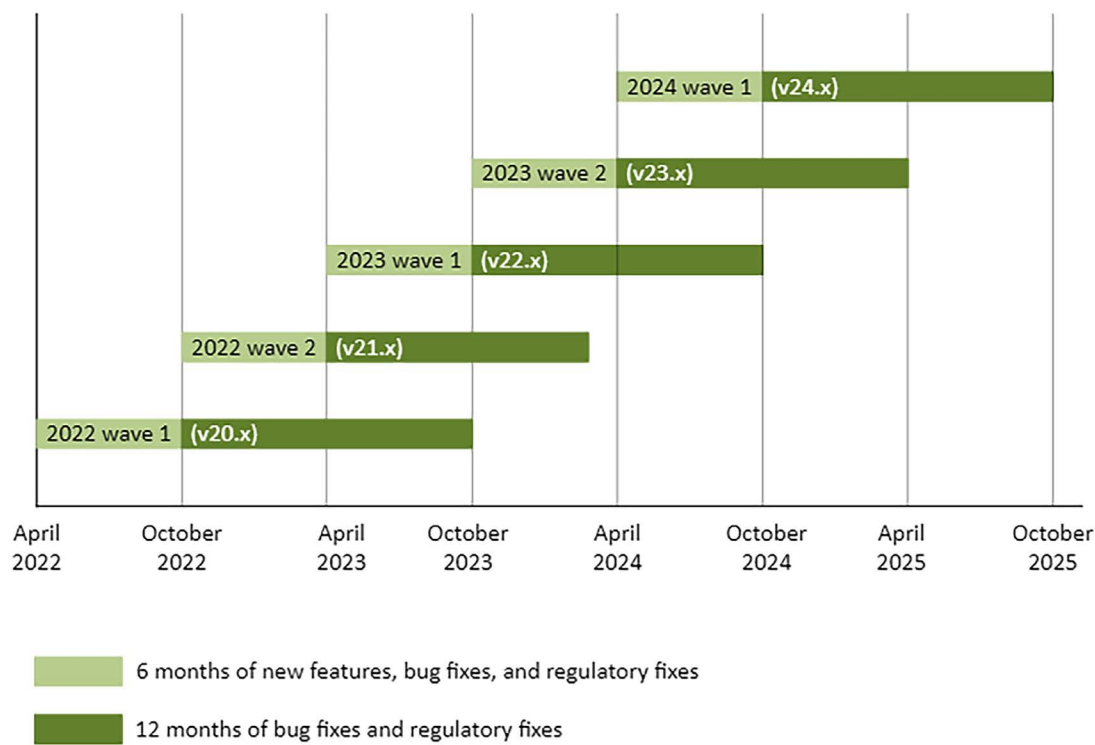


Figure 1.3: Business Central lifecycle policy

From a development perspective, this short lifecycle, with a couple of major releases per year, might be a challenge and needs to be tackled and mastered by working more efficiently than in the past when working just with the on-premises line of business. On the other hand, we should not underestimate that the continuous update represents an endless improvement for end-users, both from an application and platform perspective, and is considered a pay-off in investments.

In other words, the online world represents an opportunity to concentrate on pure development mode for partners, without the need to draft the underlying system maintenance part and its upgrade, not only in terms of hardware and software but also from a base application perspective.

To help partners concentrate on what they do best, the platform has grown a lot during these years with tremendous investments from Microsoft. This is reflected in all the different product areas, such as development, client usability, performance, telemetries, debugging tools, and so on.

Yes, this is truly the Dynamics 365 Business Central momentum.

Below is a timeline of some of what could be considered by us the best technical improvements for every specific release from October 2019 onward, considering the online version. If you would like to have a similar timeline for the application improvements, it is worth checking this blog post: *Business Central What's New in Application BC18 to BC23* at <https://blog.versionmanager.dk/business-central-whats-new-in-application-bc18-to-bc23/>

2019 Wave 2 – version 15.x (October 2019)	
<b>Development</b>	Attach and debug next
<b>Client</b>	Keyboard shortcut support
<b>Server</b>	Page background tasks (PBT)
	Application Insights telemetry for partners

2020 Wave 1 – version 16.x (April 2020)	
<b>Development</b>	Interfaces
<b>Server</b>	Feature Management to enable or test features ahead of time
	Read scale-out for resource governance

2020 Wave 2 – version 17.x (October 2020)	
<b>Development</b>	Snapshot Debugger in production
	Business Central Performance Toolkit
	Attach to a user session when debugging
	Delete extension data
<b>Server</b>	Data audit system fields are added to every table
<b>Integration</b>	Microsoft Dataverse virtual tables
	Business Central in Microsoft Teams

2021 Wave 1 – version 18.x (April 2021)	
<b>Development</b>	Report Extension
	Entitlement, PermissionSet, and PermissionSetExtension
	Add keys to base tables and table extensions
<b>Client</b>	Cloud printing using Microsoft Universal Print
<b>Server</b>	Service-to-service (S2S) authentication

2021 Wave 2 – version 19.x (October 2021)	
Development	Profiling AL performance with snapshot debugger
	Force sync of customer-specific extensions in online environments
Server	Partial records (JIT loading)
Integration	Power Automate and Power Apps connector support, finding records
2022 Wave 1 – version 20.x (April 2022)	
Development	In-client performance profiler
	AL-Go for GitHub (DevOps for partners)
Server	Report Excel layout
2022 Wave 2 – version 21.x (October 2022)	
Development	Launch in a specific company from Visual Studio Code
	DataTransfer object to enable faster data upgrades
Client	Easily switch companies across environments
	Modern Action Bar
Integration	Access Business Central with Microsoft 365 license
2023 Wave 1 – version 22.x (April 2022)	
Development	ReadIsolation
	Attach debugger to active or next session
Client	Toggle in-client analysis mode
	Actionable error messages
Integration	Copilot and Artificial Intelligence (AI) integration
2023 Wave 2 – version 23.x (October 2023)	
Development	Namespaces
Server	New Table Extension data model
	Change locking behavior to ReadCommitted (tri-state locking)
Client	More AI infusion and PromptDialog

*Table 1.2: Business Central technical development timeline*

So now, from October 2019 (and taking less than five years), Dynamics 365 Business Central demonstrates the uptake of deep integration with a lot of Microsoft Azure services and, overall, the Microsoft Power Platform and Microsoft Teams.

It has grown in its debugging experience, introducing the snapshot debugger and in-client profiling for simultaneous troubleshooting, and release after release is adding new signals to enrich telemetry data emission.

Dynamics 365 Business Central client and server runtime teams are working to improve performance at every update and provide guidelines and tools to write and execute code that is streamlined for fast data retrieval and insertion.

Last but not least: the AI hype. Since 2023 Wave 1, Microsoft started to infuse Dynamics 365 Business Central with AI and include Copilot in several business processes all over the platform and application.

This is, then, an endless endeavor that Microsoft and its partners and customers are taking to grow their business together. There are at least four main points to consider when creating an offering for a new prospect or a migration from an existing customer:

1. **Implement performance tests in advance and execute them periodically:** You might think of using the Performance Toolkit extension (<https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-performance-toolkit>) or you can opt for your own or third-party performance tools and assessments. You might also think of using both Performance Toolkit to define a customer benchmark and Waldo BCPerfTool. This last one can be forked for free in GitHub at <https://github.com/waldo1001/waldo.BCPerfTool> (with a great explanation video, 20230109 – *The less familiar performance tooling*, [https://www.youtube.com/watch?v=\\_V0hepCRrkA](https://www.youtube.com/watch?v=_V0hepCRrkA))

In the cloud, having super performant code is crucial since every customer is a “citizen” that lives in their own apartment in a big condominium. Resources are shared across the apartments, and everyone should use and care about them, to avoid starvation. Within multi-tenancy, a “noisy neighborhood” is not tolerated, in order to avoid general performance problems between tenants sharing the same cluster resources.

For this reason, it is very important to write performant code from the beginning and test load in advance and periodically.

2. **Be sure to carefully read the operational limits related to cloud deployments:** <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/operational-limits-online>.

Here are two examples of online vs on-premises scenarios to demonstrate operational limits:

- *Example one:* If you have a high number of API calls, remember that for the online version, the current maximum per minute is set to 6000 per user / minute. If you have a high volume, you need to implement retry routines to avoid HTTP 429 errors (too many requests).

Rate	The maximum number of OData V4 requests that can be submitted within a 5-minute sliding window. When this limit is exceeded, an HTTP response code 429 - Too Many Requests is returned. The more users you have in your environment, the more requests you can submit to it around the same time, as long as we can continuously scale our resources. If many requests are being submitted around the same time and we couldn't sufficiently scale our resources, you might experience throttling in submitting your requests.	6000, see frequently asked questions on per-user limits.
------	--	--

Figure 1.4: Operational limits in the online version of Business Central

- *Example two:* If you have a lot of companies within the database, these could impact administration activities such as point in time restore, creating and downloading BACPAC files, and even CI/CD deployments. For these and other reasons, Microsoft limited the number of companies to be created in SaaS to 300.

You can read more about the operational challenges with the online version when having a database with many companies here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/environment-company-limit>.

All in all, be sure to acknowledge a customer’s business processes, volumes, and technical and business growth rate to avoid any unpleasant surprises when going live.

3. **24/7 business considerations:** If your customer has a tight and scheduled downtime window, like typical 24/7 businesses, you should be aware that SaaS deployment involves a minor update every month and a major update every 6 months. These could be tested in advance in a sandbox and planned at will.

But keep in mind that Microsoft is still deploying applications and platform hotfixes through the pipeline when it is more convenient or generally needed. This might even happen every day in a week, even though Microsoft is trying to limit them as much as possible and cannot be controlled by partners or a tenant being kept outside pipelines.

Such online service operations and their cadence are described officially here:


<https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/service-overview#service-operations>.





See the sentence “Service operations happen all day, every day, to always provide [the] best experience.” Point taken.

4. **Storage pricing considerations and archiving data:** When purchasing Essential or Premium licenses, these go in a bundle with a fixed quantity of storage for all the environment databases. Typically, a Dynamics 365 Business Central environment starts with a capacity of 80 GB.

If your customer is performing a lot of transactions in short periods of time, your database growth might skyrocket and exceed the initially purchased database capacity.

Customers could purchase additional storage space as an additional license with a monthly-paid extra cost. Below, you can see the latest update from July 2021 in relation to storage costs:


**Dynamics 365 Business Central**  
 Storage changes effective July 1

	Current	New storage plan (starting July 1, 2021)	
 Base entitlement	80 GB	80 GB	
 Per user allocation	None	2 GB/User (Essentials) 3 GB/User (Premium)	
 Additional GB	\$40	\$10	
 Additional 100 GB	\$4,000	\$500 (5\$/GB)	

Note: Storage is inclusive of all BC Online Customer Data ('Relational' and 'Blob') both in current and new storage plan

Figure 1.5: Storage costs

Be careful, then, in calculating the growth of the database over time, and be fully aware that this could most likely be an important economic factor in deciding whether to go online or still deploy your customer on-premises. In both cases, it would always be good practice to keep monitoring database capacity and deciding from the beginning where to store and retrieve complex data (such as, files, images, videos, and so on) and implement a “delete or archive” business process.

Finally, be sure to always be up to date on the latest pricing for Dynamics 365 Business Central licensing since there might be refinements or periodical price adjustments that might give you a different profitability comparing SaaS with on-premises. The link below will let you download the latest updated licensing guide: <https://go.microsoft.com/fwlink/?LinkId=866544>.

Now that we have just emphasized and highlighted the Dynamics 365 Business Central online momentum, product growth, and some retrospectives considering SaaS vs on-premises deployments, let's see the expression of the modern development by unveiling the *Universal Code initiative* and how this will be a game changer.

## The Universal Code initiative

The word “initiative” suggests that this is a good practice encouraged in the development community. In this specific case, Microsoft has made this encouragement even more attractive by adding a fee to on-premises extensions that are not adhering to this restrictive cloud scope. Writing universal code extensions means writing code targeted for ubiquitous deployment types: cloud and on-premises. A synonym of universal code is *cloud-optimized*.

The main benefits that drive this initiative are:

1. Standardizes a common partner development target with a cloud-first approach.
2. New or existing customers could purchase the same AppSource extension, without the need for any extra legacy dependency.
3. Faster upgrades, updates, and cloud migrations.
4. Efficiently uptakes the latest and greatest Microsoft technologies.
5. Replaces outdated and old-fashioned partner programs like the **Registered Solution Program (RSP)** and the **Certified for Microsoft Dynamics Program (CfMD)**.

From a technical perspective, this simply implies editing the extension manifest file (`app.json`) using the well-known property `"target": "Cloud"`. But this is not as easy as changing one single property.

Every developer, no matter if a newbie or otherwise, knows that changing the target parameters has a significant impact when compiling extensions. The most common problems are:

1. It is not possible to reference dotnet objects.
2. It is not possible to reference functions with scope on-premises.
3. Code written to access local resources (for example, files or directories) must be refactored.
4. Local printers cannot be used for server-side printing.

Those are the principal obstacles that developers will find in their everyday life when attempting to make their code universal. Statistics from January 2022 found the following percentage breakdown on what might be the showstoppers in writing universal code and taking the leap to the cloud:

- **20%:** Lack of knowledge of serverless architecture
- **18%:** Application code performance
- **17%:** Unpredictable costs due to a different development approach
- **16%:** Interface with production machines and automated warehouses
- **13%:** File management
- **10%:** Printing
- **6%:** Lack of events in first-party extensions (mainly base application)

More information can be found at <https://demiliani.com/2022/01/12/business-central-universal-code-initiative-are-you-really-so-surprised/>. As you might deduce, the impact could not only be a technical one but also have implications for entire business processes if you are migrating an existing customer from on-premises.

What if you continue to write code with a scope different than the cloud for on-premises deployments? As mentioned earlier, this is not technically prohibited, but Microsoft is applying growing extra fees for non-universal code customization and extensions, which started in January 2023. The extra fees are applied if the code implemented by a partner has the following characteristics:

- **It is not in extensions** (the standard application has been customized using C/AL)
- **It is not cloud-optimized (universal)** (partner application code is isolated in extensions but compiled using an on-premises target)

The following table displays the current prospective timeline (NOTE – fees might be subject to changes):

Recurring fee(s) timeline	Implemented code is not in extensions	Implemented code is not cloud-optimized
2023	\$75	\$0
2024	\$125	\$75
2025	\$250	\$175
2026 onward	TBD	TBD

Table 1.3: Prospective fees timeline

If the economic reason is not enough to change your mind or that of your co-workers, you might think of this initiative in other terms.

The Dynamics 365 Business Central team will introduce more and more features targeted for the online version or natively integrated with it, and Microsoft will continue to invest billions in its cloud platform and services.

Just do the math, then, and decide your mid- to long-term strategy to take the leap to the cloud.

You can read more about the Universal Code initiative in these blog posts:

- *The Dynamics 365 Business Central Universal Code initiative is live:* <https://cloudblogs.microsoft.com/dynamics365/it/2022/10/28/the-dynamics-365-business-central-universal-code-initiative-is-live/>
- The “Universal Code Initiative”: <https://www.waldo.be/2021/12/23/microsoft-dynamics-365-business-central-universal-code-initiative/>
- *Business Central Universal Code initiative: are you really so surprised?:* <https://demiliani.com/2022/01/12/business-central-universal-code-initiative-are-you-really-so-surprised/>

And if you are working for a partner, you can also double-check the official documentation in Partner Portal at <http://aka.ms/BCUniversalCode>.

Now that we know that there is hard work to do to refactor old legacy code into modern universal code, let us have a look now at how the Dynamics 365 Business Central evolution momentum is also driven by the community and its open-source contribution.

## The role of open source and social networks

To understand the role of open source in Microsoft, we need to consider two things about the environment we are working in.

When implementing any Microsoft product or, effectively, any classic software for whatever purpose, you have official instructions to follow. This is generally true for heavily standardized software deployment types with high repeatability. That said, with any ERP implementation, you should still take care of the following:

- Hardware and software requirements
- Business needs and peculiarities of processes for a specific customer

It is possible that you will need open-source material to meet these requirements.



Luckily, hardware and software requirements are simpler within SaaS compared to an on-premises version. Despite this, the complexity of implementation remains (due to the infinite combination of business needs, processes, and technical solutions).

Another point to keep in mind is that an ERP system is part of a fast evolutionary process due to the high dynamism of the society we live in. Today's requirements might not be needed tomorrow. Just consider the FAX no. field – is this a needed field anymore? There will surely be fields that are required today that will not be in the future.

Another example of evolution driven by society is the evolution of security protocol and authentication. In a few years, most probably, no one will authenticate for online services using basic authentication such as a user ID and password in isolation. This is driven by our society requiring more and more security measures to protect data. (Note that Dynamics 365 Business Central has deprecated Business Authentication for a while now, in favor of a secure, robust, and worldwide accepted OAuth 2.0 protocol.)

These are just a couple of examples of how dynamic an ERP evolution could be and how complex and diverse its implementation and management are.

It is the society that we are living in that dictates both pace and trends. Whoever is adapting faster to society's evolution rate and its new requirements will be in a dominant position compared to others.

For this reason, Microsoft increases its resiliency to this evolution by adopting a practical open-source approach. Combined with the normal Microsoft support, partners (and customers) can now contribute to the evolution of ERP in multiple ways. There are preferred communication channels, depending on the area of interest, that can be used for free to communicate with the Dynamics 365 Business Central product group or share questions, concerns, and ideas. The following list outlines the most important channels:

- <https://aka.ms/bclinks> : If you want to discover all the aka.ms links related to Dynamics 365 Business Central and what they are for.
- <https://aka.ms/bcideas>: Suggestions for new features and enhancing existing features and capabilities. Directly creates an internal record for Dynamics 365 Business Central's engineering DevOps. This elevates prioritization and rankings for a specific feature implementation request. You could also comment on and/or vote for existing suggestions.

- <https://github.com/microsoft/ALAppExtensions/issues>: Extensibility-related requests. File requests for new integration events, extensibility bugs, conversion from option to enum, function exposure, and scope changes here. If accepted, changes are typically implemented in one or two of the next minor releases or in the next major one.



*NOTE: This site will transition over time into <https://github.com/microsoft/BCApps>, which will be the future collector for all open-source changes to the application. Currently it's hosting the master branch (the next release version: 24) for System Application and side features such as Test Framework and Performance Toolkit.*

- <https://github.com/microsoft/al/issues>: Bugs found in the AL extension for Visual Studio Code. In accordance to with its README file, it should be strictly used to report bugs in the insider or beta version (internally at Microsoft called **Master** or **vNext**). For errors or inconsistencies in versions in mainstream support, the typical Microsoft support channel through CSP partners must be used.
- <https://learn.microsoft.com/en-us/collaborate/>: Bugs found in application or platform preview versions. Collaborate is restricted and accessible only for **Independent Software Vendor (ISV)** or **Value Added Resellers (VAR)** partners.
- <https://www.yammer.com/dynamicsnavdev>: Direct discussions with the product group and other partners. Registration to participate is restricted and needs to be internally approved by Microsoft. It is divided into different groups such as announcements, Power Platform, AppSource, Development, etc.
- [https://twitter.com/search?q=bcalhelp&src=typed\\_query](https://twitter.com/search?q=bcalhelp&src=typed_query): To stay up to date with development-based (and non-development-based) requests, questions, and polls. Using #Bcalhelp, you can ask immediate questions and get an answer quite quickly, or you can use X (formerly known as Twitter) as a networking channel. Also, #Msdyn365bc is another favorite tag to follow or post.
- <https://community.dynamics.com/forums/thread/?partialUrl=business>: The official Dynamics 365 Business Central Community. It could be considered as the Stack Overflow for Dynamics 365 Business Central where everyone can post questions, answers, or create useful blogs. The community is moderated directly by Microsoft, and you might find useful tips, tricks, and answers from certified **Most Valuable Professionals (MVPs)**.

There might be a vast plethora of blogs or websites where you could find the information that you need to resolve a specific problem or unblock a development task of your current project but the list reported above is a valuable starting point to find out what you need or what you will be in need of from Dynamics 365 Business Central.

And if you do not have the time to browse all these blogs or you're too lazy for it, no problem. One of the most recognized members of the community, Dmitry Katson, created an **Artificial Intelligence (AI)** powered search engine that has indexed (almost) all the information related to Dynamics 365 Business Central. Whatever inquiries you might have, just visit <https://www.centralq.ai/> and type them in.



CentralQ Chat is live on AppSource! Learn more...

# CentralQ

AI search, fueled by the [collective knowledge](#) of Business Central community.

**Focus on**

☒
AppSource

☐
System App

Figure 1.6: Using CentralQ for Business Central queries

But this is not enough. A real open-source philosophy implies letting everyone *directly* contribute to the product development rather than indirectly filing specific requests for review or implementation.

These days, it is possible to directly create a **Pull Request (PR)** to all Microsoft standard extensions (so-called *first-party* extensions) except Base Application, using this GitHub repo: <https://github.com/microsoft/ALAppExtensions>.



**NOTE:** This site will transition over time into <https://github.com/microsoft/BCApps>, which will be the future collector for all open-source changes to the application. Currently it's hosting the master branch (the next release version: 24) for System Application and side features such as Test Framework and Performance Toolkit.

This GitHub repo is divided into *modules*, which store system applications, and *apps*, which collect localized applications and all Microsoft feature and utility extensions, such as Universal Print, Email Logging, and Data Search. In the **Pull Requests** section, you can clearly see the current contribution and Microsoft reviewing and approval activities.

If you have found a bug in the current version or want a specific feature to be enriched or enhanced, you could follow the official documentation to change an existing module or even create a new one: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-change-a-module>.

You could also check out this video from Erik Hougaard (MVP), *How to submit a pull request to Microsoft for Business Central*: <https://www.youtube.com/watch?v=3Q9ikdBYJvQ>.

What about Base Application? For several years, it has been kept out of open source for several reasons. With Dynamics 365 Business Central 2022 Wave 2 (version 21.x), it has been finally announced that Base Application will also be open to contributions by selected representatives from the community in this repo: <https://github.com/microsoft/BusinessCentralApps>.

If you want to know more about how to get involved with Business Central open source, this video from Microsoft representatives, *BC TechDays 2022 – Collaborating on Business Central's open source platform*, will provide all the needed insights: <https://www.youtube.com/watch?v=X10yTEpTmfw>.

And that's a checkmate from Microsoft in going all-in with open-source applications. It is trivial to say that the more you contribute, the more chances you will have to also get rewarded by Microsoft as an MVP or simply be recognized as a community influencer.

Packing up all the information, it would be helpful for you to know, as a developer, what will come in the future of Dynamics 365 Business Central and what skills will be needed to tame the beast.

## The future perspective

Microsoft Azure and Microsoft 365 are now solid and mature and have considerable **Returns On Investment (ROIs)** for customers.

They are among the best sources of revenue for Microsoft and are where the most investment and capital are redirected. For the last decade, all Microsoft products have been requested to align with Microsoft's strategy of increasing the consumption of these two flagship services.

Dynamics 365 Business Central was designed to perfectly fit into Microsoft's strategy: it brings new potential SMB ERP customers into this offering to accelerate the top-class Microsoft cloud service consumption and its growth.

Dynamics NAV and Dynamics GP existing customers are warmly encouraged by Microsoft to subscribe to Dynamics 365 Business Central SaaS Essential or Premium tier, instead of receiving a typical offering for on-premises deployment or an upgrade renewal.

A huge product transformation to drive online adoption has been announced at the latest Microsoft and non-Microsoft events and, as we have just learned, the Universal Code initiative is a real and tangible demonstration of this long-term strategy.

The following is the current roadmap for the product that was recently presented in one of the latest **Directions EMEA** (<https://directions4partners.com/>) events delivered all over the globe.

# Dynamics 365 Business Central

## 2024 release wave 1 investment areas

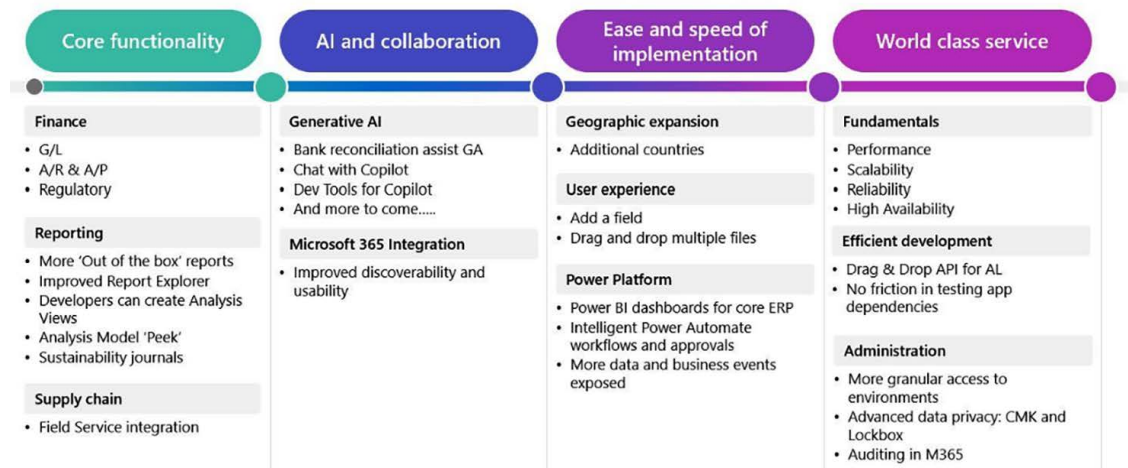


Figure 1.7: Business Central product roadmap

Based on the current trend and roadmap, we suggest that Dynamics 365 Business Central developers acquire skills related to the following topics:

- Visual Studio Code and AL Language
- PowerShell
- Git
- GitHub and Azure DevOps
- JavaScript and web-based development
- AI and Machine Learning techniques
- Azure services for developers (such as Azure Functions and Cognitive Service.)

Developers, integrators, and architects should also become familiar with the following:

- Docker
- Azure compute services (such as Azure VMs and Azure Storage)
- Microsoft 365 services and Dynamics 365
- Dataverse
- Power Platform



It is highly recommended for partners to subscribe to the *Dynamics 365 Business Central Ready to Go* program and benefit from its endless and constantly updated learning catalog. You can read more about it at <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/readiness/readiness-ready-to-go?tabs=learning>.

Considering the existing customers that still are on-premises with Dynamics NAV and Dynamics GP, the best suggestions we can provide for customer and partner organizations to embrace the Universal Code initiative are as follows:

- Move all the existing private IPs outside standard code with event-driven development as much as possible. If this requires new events in the standard application, request them through the appropriate Microsoft channel.
- Whatever can be isolated into event-driven development can also be packaged as an extension. Move as many private IPs as possible into a refactored universal extension.
- Refactor all code to make it work seamlessly in the web client and focus all skills on universal code development.
- Train all your salespeople, installers, developers, functional application experts, and everyone else who's using or demoing the web client. Live and breathe using Dynamics 365 Business Central modern clients.
- Collaborate in GitHub and use social media.
- Stay up to date on X (formerly known as Twitter), LinkedIn, and Yammer (for partners). Take note of your own business process showstopper and share it with the Dynamics 365 Business Central community and development team by actively participating in official and unofficial forums and dedicated product events all around the world.
- Get used to, or even deeply specialize in, modern Microsoft and non-Microsoft technologies.

Last tip: take an official Microsoft certification.

No matter if you are in the Dynamics 365 Business Central business as a developer, installer, sales manager, or whatever role, the milestone exam *MB-800: Microsoft Dynamics 365 Business Central Functional Consultant* (<https://learn.microsoft.com/en-us/certifications/exams/mb-800>) should be a must for everyone in your organization.

If you are a developer or a freelancer, you could also think of enlarging your certificate portfolio by taking several *fundamentals* exams like Azure fundamentals and Power Platform fundamentals ([https://learn.microsoft.com/en-us/certifications/browse/?resource\\_type=certification&type=fundamentals](https://learn.microsoft.com/en-us/certifications/browse/?resource_type=certification&type=fundamentals)).

These exams will help you and your company grow and be familiar with the latest Microsoft technologies and guarantee a smooth integration with Dynamics 365 Business Central and faster knowledge uptake.

Also, do not forget that a new exam, MB-820, has been finally announced as the associate certification for developers. The Beta exam started in January 2024. This new exam is an *Intermediate* level exam, at the same level of MB-800 for the Functional Consultant and it is a must for all Dynamics 365 Business Central developers.

## Summary

We have now completed an overview of what is available in – and what will be in the future of – Dynamics 365 Business Central.

We have focused on Dynamics 365 Business Central by considering it as a product in evolution. This will be beneficial for you when it comes to understanding what the product can offer in terms of localizations, new features, and how it targets the SMB market segment for your projects.

We have covered the Universal Code initiative and how you can contribute to the Dynamics 365 Business Central ecosystem. The important thing is for you to be prepared for what might happen in the future and have the skills you will need in order to evolve together with this fascinating product.

In the next two chapters, we will take our first steps using Visual Studio Code as a development environment and coding with some basic instructions. Spoiler alert: if you are an experienced developer, these next two chapters could be more of a recap or refresher for you. For newer developers, you will learn about some vital fundamentals.

## Leave a review!

*Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.*



*\*Limited Offer*



# 2

## Mastering a Modern Development Environment

In the previous chapter, we introduced Dynamics 365 Business Central and the amazing momentum that its online version is enjoying.

In this chapter, we will take a close look at the development environment. We will discuss the main shortcuts, tips, and tricks related to Visual Studio Code, the official development platform, and the AL Language extension, the development language extension. The union between Visual Studio Code and AL defines the so-called modern development environment.

AL Language is the official Visual Studio Code extension provided by Microsoft, free of charge, through the online marketplace. AL stands for **Application Language**, although it's typically referred to as AL Language. Officially released for the first time back in 2017 to extend what was then called Dynamics 365 for Financials (formerly Project Madeira), it is now a solid, fully fledged development language that extends Dynamics 365 Business Central. It comes equipped with a lot of features that greatly enhance developers' productivity and coding quality.

The main goal of this chapter is to help Dynamics 365 Business Central developers understand what the development platform offers, unleash their potential, and become proficient in their daily coding activities.

In this chapter, you will learn about the following:

- What the Visual Studio Code user interface is composed of, and the purpose of each section
- How to be proficient in using the most powerful Visual Studio Code editing features
- What the AL Language extension is and what it consists of
- Turning on code analyzers to check and enforce encoding rules
- Enabling GitHub Copilot to accelerate the writing of code

## The Visual Studio Code user interface

Visual Studio Code is one of the most widely used development environments worldwide. It is engineered to make it easy and quick to design cloud- and web-based applications, using a vast plethora of extensible languages. The application is focused on maximizing code editing and unleashing the developer's potential by providing useful shortcuts to provide quick access to all that is needed in a specific development context.

Visual Studio Code can be downloaded officially at <https://code.visualstudio.com/> and, like most **Integrated Development Environments (IDEs)** nowadays, it has two different branches or deployment channels for its build: stable and insider. Together with these, it could also be used as a web application (<https://vscode.dev/>) or deployed on different platforms than Microsoft Windows, like iOS or Linux.

When you start Visual Studio Code, freshly installed locally, it will show you the typical Welcome page:

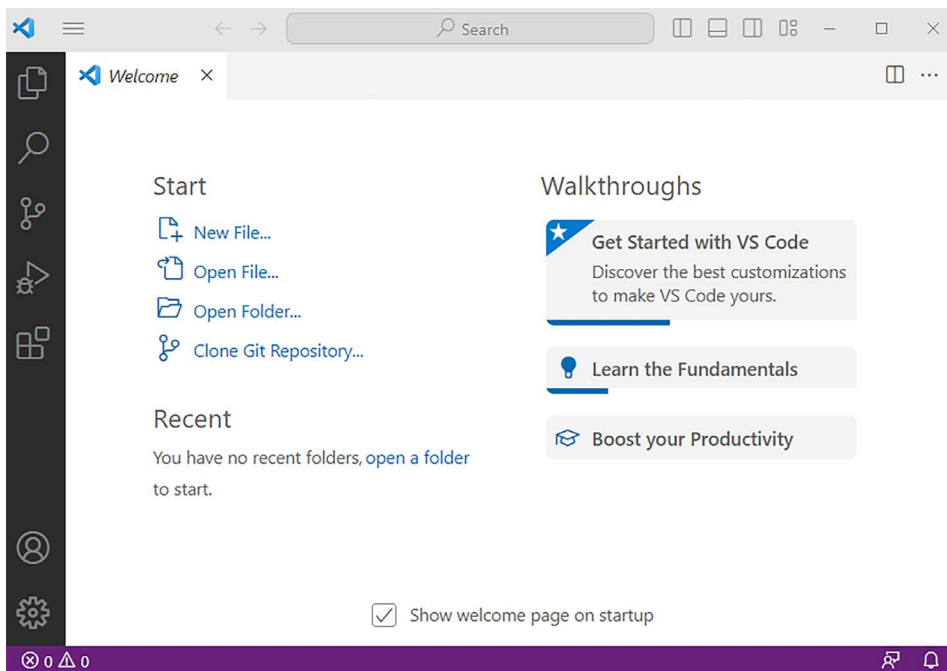


Figure 2.1: Visual Studio Code Welcome page

The Welcome page contains the following:

- **Start:** Shortcuts for creating and opening files and folders
- **Recent:** A list of recently opened files and folders
- **Walkthroughs:** A series of sliders that would help you tailor Visual Studio Code through your favorite settings such as background color themes, useful features enablement, and so on

The Welcome page is loaded whenever you run Visual Studio Code as a new window (**Ctrl + Shift + N**). It is possible to change this behavior by unchecking **Show welcome page on startup** or clicking **File | Preferences | Settings** and searching for **Welcome page**.

The Visual Studio Code environment is divided into five main areas:

- Code editor
- Status bar
- Activity bar
- Sidebar
- Panels area

Let's look at each of them in the following sections, but if you want to check the documentation reference first, please visit <https://code.visualstudio.com/docs/getstarted/userinterface>.

## Code editor

The code editor is where you write your code and where you spend most of your time. It is activated when creating a new file, or when opening an existing file or folder.

You can edit just one single file, or you can even load and work with multiple files at the same time, side by side, in editor groups:

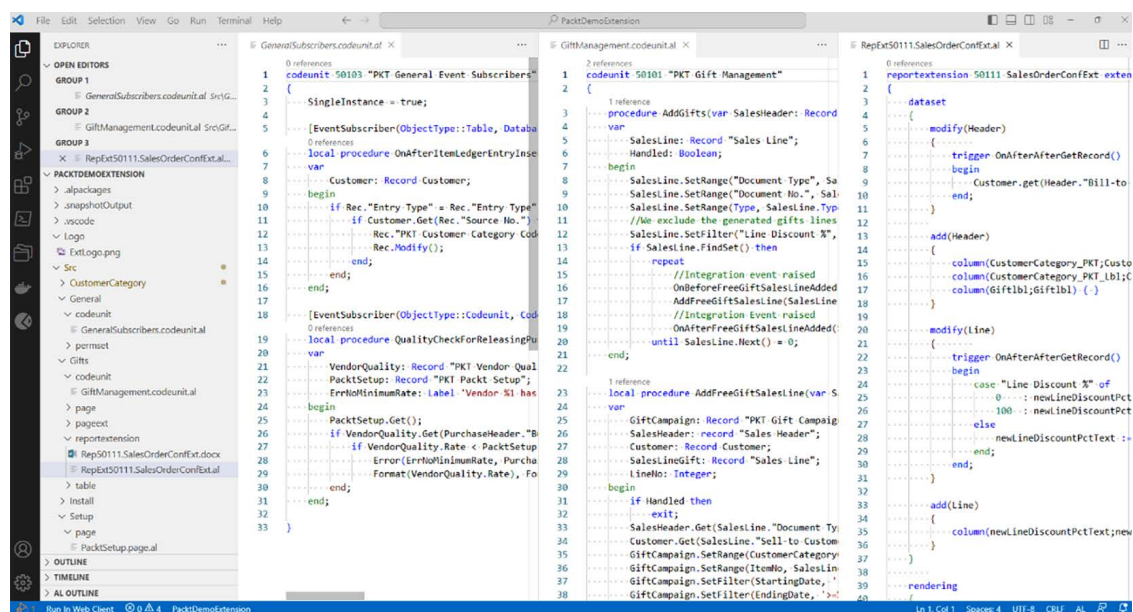


Figure 2.2: Working on multiple files with editor groups

There are different ways to have multiple file views; three are mentioned here:

- Select a filename in the **EXPLORER** bar, then right-click and select **Open to the Side** (*Ctrl + Enter*).
- Press *Ctrl* + click on a filename in the **EXPLORER** bar.
- Press *Ctrl* + \ to split the editor into two parts.

Editor groups will accommodate several files, dividing the space equally between them. You can move through the different file editors by simply pressing *Ctrl + 1*, *Ctrl + 2*, *Ctrl + 3*, ..., *Ctrl + N*.

Editor windows can be resized, reordered, and zoomed in/out according to your needs. To zoom in/out, press `Ctrl + +/Ctrl + -`, or **View | Zoom in/Zoom out**.



Zooming applies to all Visual Studio Code areas, not only to the code editor.

Visual Studio Code also provides an easy way of navigating between files with shortcuts. The quickest way is to press `Ctrl + Tab`. This will open the list of files that have been opened since Visual Studio Code started.

Last but not least, `Ctrl + R` could be considered the most used one. This will open a list of the recent files opened in the Command Palette.

## Status bar

The status bar typically contains information about the currently selected file or folder. It also provides some actionable shortcuts:



Figure 2.3: Status bar shortcuts

From left to right, the status bar contains the following information:

- If Git is enabled, it will report version control information, such as, for example, the current branch.
- Number of errors and/or warnings detected in the current code.
- Cursor position (line position and column position).
- Indentation size and type (spaces or tabs).
- Encoding of the currently selected file.
- Line terminator: **Carriage return (CR)** and/or **line feed (LF)**.
- Language used to process the code in the selected file. If you click on the language, a menu list will appear, and you should be able to change the processing programming language.
- Feedback button, which you can use to share your feedback about Visual Studio Code on Twitter/X or open a bug or feature implementation request.
- Notification icon: This shows the number of new notifications, which are typically related to product updates.

The status bar has a conventional colorization, and it changes depending on what's processing. For example, it is purple when opening a new window or a new file, blue when opening a folder, orange when debugging, and so on.

## Activity bar

This is on the left side of the workspace and contains shortcuts to the sidebar. If a shortcut is clicked, the sidebar that belongs to the tool that has been chosen becomes visible. Clicking again, or – for the shortcut lovers – pressing *Ctrl + B*, makes it disappear.

## Manage

The **Manage** button is shown with a gear icon at the very bottom of the activity bar. If you click on it, a pop-up menu with a list of commands appears. These commands are used to tailor and customize Visual Studio Code at will, or to search for updates.

## Command Palette

The Command Palette is one of the most important tools in Visual Studio Code. Its purpose is to give quick access to standard and extended commands. There are different ways to run the Command Palette:

- *F1* (most used by all developers)
- Keyboard shortcut: *Ctrl + Shift + P*
- **View | Command Palette**
- The **Manage** button (the gear icon) | **Command Palette**

The Command Palette is not only good for showing menu commands but it can also perform other actions, such as installing extensions. You can browse through it to review the huge list of available commands. Commands are indexed and searchable. Just type a few letters to get a filtered list. It's worth mentioning the long list of keyboard shortcuts that are available for most of these commands.

One very important thing to know about the Command Palette is the use of the *>* sign. When you press *Ctrl + Shift + P*, the Command Palette pops up with the *>* sign and shows the list of commands available. Take a look at the following screenshot:

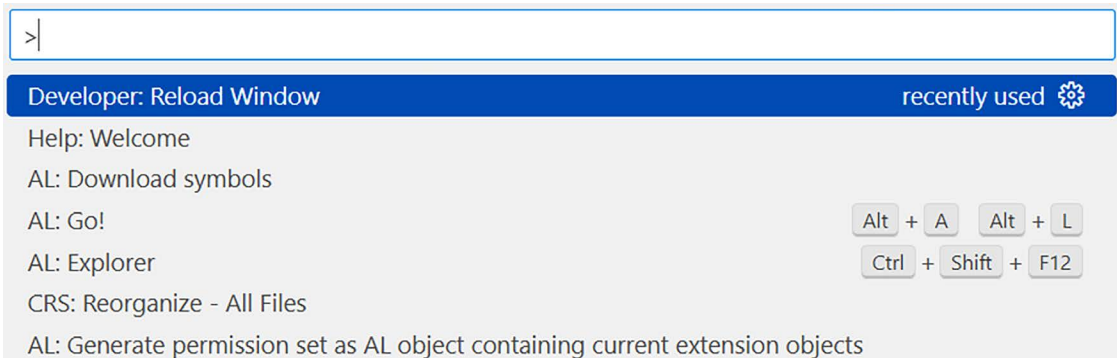


Figure 2.4: Commands suggested by the Command Palette

If you remove the > symbol, Visual Studio Code uses the Command Palette to show a list of the recently opened files. The following screenshot shows this:

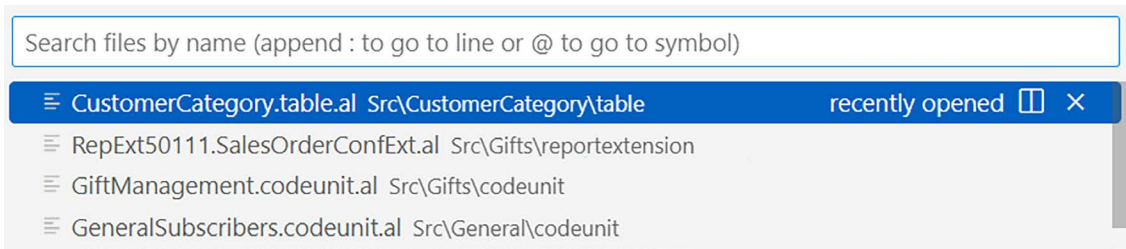


Figure 2.5: Recent files in the Command Palette

The power of this feature is that, without using the mouse, you can open the Command Palette, run a command, remove the > character, and select a file to edit (which is equivalent to using the shortcut *Ctrl + P*). That's fantastic for development productivity.

## Sidebar

The sidebar is the place where you will interact the most with the code editor. It is context-sensitive, and you will find five standard activities, each enabled by the corresponding icon in the view bar.

### EXPLORER (Ctrl + Shift + E)

EXPLORER provides a structured and organized view of the folder and files that you are currently working with. The OPEN EDITORS sub-view contains the list of active files in the code editor. Below this section, there is another sub-view with the name of the folder that is open:

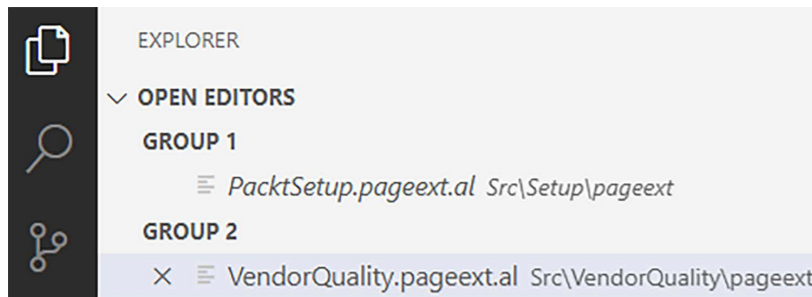


Figure 2.6: OPEN EDITORS sub-view

If you hover over the OPEN EDITORS sub-view, four action buttons will be shown: **New** (*Ctrl + N*), **Toggle Vertical/Horizontal Editor Layout** (*Shift + Alt + O*), **Save All** (*Ctrl + K + S*), and **Close All Files** (*Ctrl + K* or *Ctrl + W*):

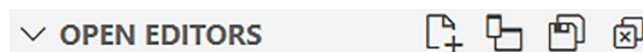


Figure 2.7: OPEN EDITORS actions

Hovering over the folder name or any part of the sidebar (in this example, **PACKTDEMOEXTENSION**) will make four action buttons visible:



Figure 2.8: Example folder action buttons

From left to right, these are **New File**, **New Folder**, **Refresh**, and **Collapse All**.

Right-clicking on a folder or filename will open a context menu that shows common commands such as **Reveal in Explorer** (*Shift + Alt + R*), which opens the folder that contains the selected file. You can also copy the file path via **Copy Path** (*Shift + Alt + C*).



When you start working with it, you might not like the **OPEN EDITORS** area, especially when you are in the middle of heavy development, since this takes up quite a lot of your screen when you have lots of files open at the same time. You can easily disable Open Editors to get that space back by clicking on the **EXPLORER** bar ellipsis and unchecking **Open Editors**.

Further down in the **EXPLORER** bar, there is another section called **OUTLINE**. It gives a very useful tree view of members and types for a specific file, based on its language extension. If you have AL Language already installed, you might have this enabled. Take a look at the following screenshot:

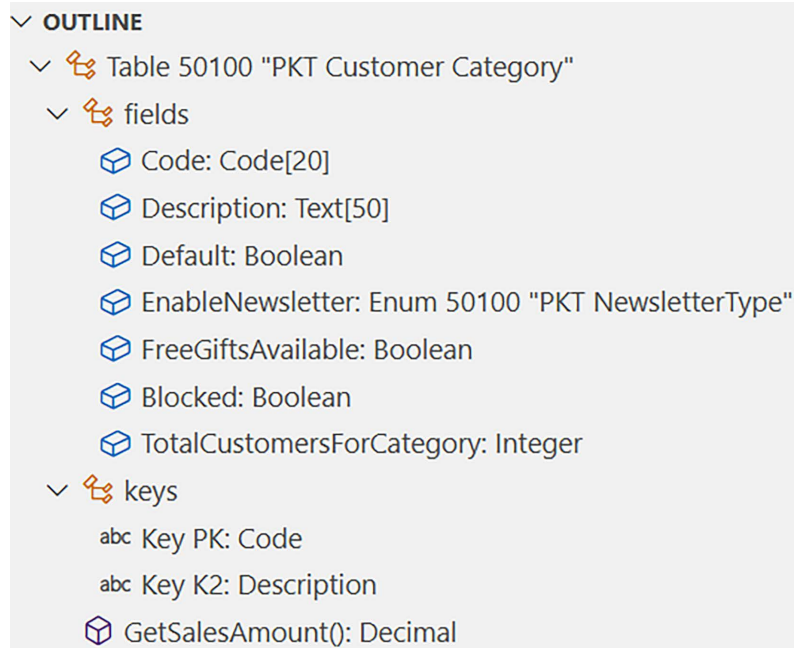


Figure 2.9: OUTLINE tree view

This is a proficient option when you are developing complex objects and you want to jump into a specific area with just one click.

## SEARCH (Ctrl + Shift + F)

This is a powerful tool for searching for and replacing text in files. It is possible to opt for a simple search with one or more keywords, and you can use wildcards such as `*` and `?`. Alternatively, you can opt for creating a complex search based on **regular expressions (regexes)**. There are also advanced options to include and/or exclude files or file types.



If you would like to learn more about regexes in Visual Studio Code, the following video is a must-watch, *20230320 - RegEx - basic and advanced scenarios*: <https://www.youtube.com/watch?v=8qEHfD-QkXk>.

The **SEARCH** action bar is helpful for developers when searching the **Where used** field or variables in all files within an extension folder. Take a look at the following screenshot:

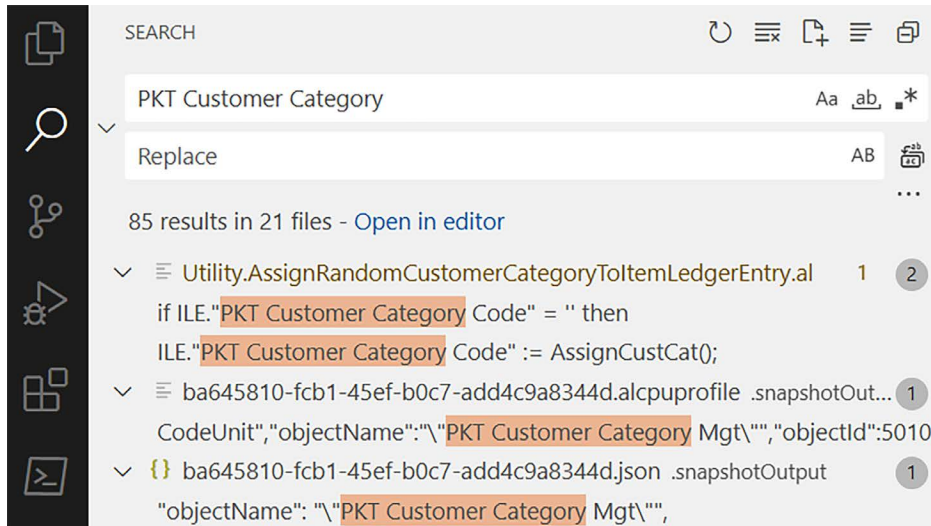


Figure 2.10: Searching extension files

Search outputs are listed in a tree view that lists all files containing the search keyword and shows a small piece of code related to the line that they belong to in the file. The keyword match is highlighted in the tree-view, as well as within the code editor. These can all be collapsed by clicking the **Collapse All** button.

It is possible to reset the search results by clicking the *Clear Search Results* button. It is also possible to turn sensitivity on/off and switch to regex view.

## SOURCE CONTROL (Ctrl + Shift + G)

Visual Studio Code provides native integration with one of the most widely known source control management systems: Git. The basics and integration of Git will be discussed in *Chapter 15, DevOps for Business Central*.

## DEBUG (Ctrl + Shift + D)

Visual Studio Code is not just a code editor for editing flat files. It also ships with an out-of-the-box integrated debugger framework that can be extended to debug different platforms and languages.

Visual Studio Code does not provide any debugging capability for Dynamics 365 Business Central per se. This comes embedded in the AL Language extension for Visual Studio Code, which extends the existing .NET core debugger. In *Chapter 10, Debugging*, we will discuss this argument in great detail.

## EXTENSIONS (Ctrl + Shift + X)

The EXTENSIONS sidebar is used to browse the online marketplace for extensions for Visual Studio Code, which includes an ever-growing number of additional languages, debuggers, tools, helpers, and much more. AL Language is an extension for Visual Studio Code developed, maintained, and published by Microsoft. In the Visual Studio Code marketplace, you can also download several helpful extensions that extend (extensions for an extension) the AL Language extension and help Dynamics 365 Business Central developers be more efficient and productive and write code professionally and more proficiently. Take a look at the following screenshot, which shows typical Visual Studio Code extensions installed for Dynamics 365 Business Central:

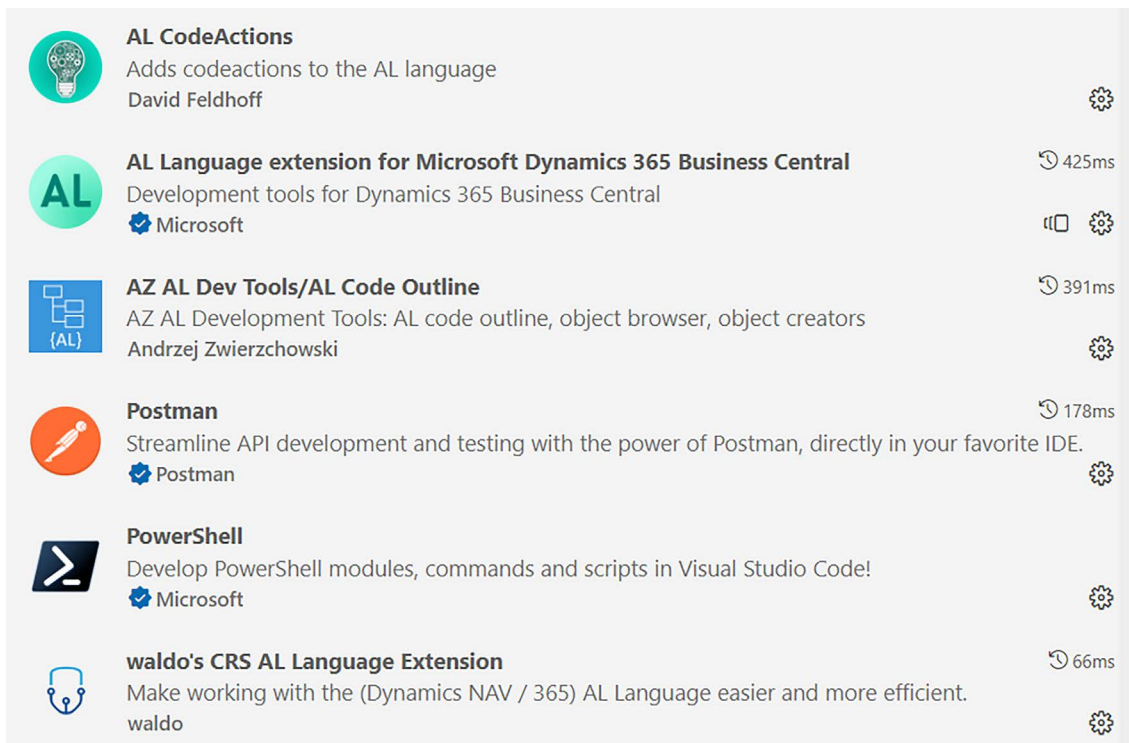


Figure 2.11: Extensions in Business Central

In the EXTENSIONS bar, it is possible to search the online marketplace or install an extension manually. You can also see the list of installed, outdated, recommended, and disabled extensions and sort them according to different criteria.



Some extension packages are meant to download and install a set of extensions. With Dynamics 365 Business Central, you might think of downloading and installing AL Extension Pack from <https://marketplace.visualstudio.com/items?itemName=waldo.al-extension-pack> or SD Extension Pack for Dynamics 365 Business Central from <https://marketplace.visualstudio.com/items?itemName=StefanoDemiliani.sd-extpack-d365bc>.

It is also possible to perform actions on a single extension by right-clicking on it. An extension can be enabled, disabled, disabled per workspace (a workspace could be a project or a folder), and so on. There are also a couple of cool features related to Visual Studio Code extension deployments that are worth mentioning. The first is the ability to install another version of the extension (for example, click on the gear icon in the AL Language extension and choose **Install Another Version...**):

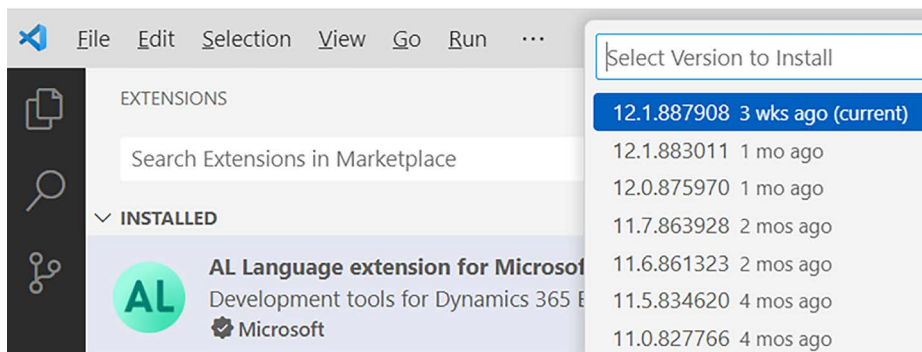


Figure 2.12: Installing extension versions

The second is the ability to deploy a pre-release instead of the official version. See below the button to deploy the pre-release for the AL Language extension:

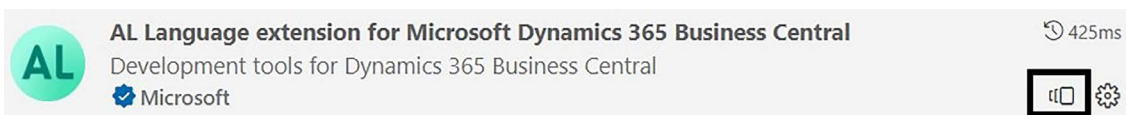


Figure 2.13: Deploying a pre-release

This is very useful for Dynamics 365 Business Central developers, in case there are regression behaviors or bugs in higher-AL Language extension versions or the official release version. This is also useful when developments target a specific platform version.

## Panels area

Visual Studio Code not only shows detailed analysis and information related to your code but also has access and display information coming from other sources such as Git, installed extensions, and debuggers. These outputs are logged into the Panels area, which, by default, is at the bottom, but could be easily moved to one side of the workspace using the **Move Panel Right** button, enabled by right-clicking on the panel's title bar. It is possible to restore the original layout with the **Move Panel to Bottom** button, or even **Hide Panel** (*Ctrl + J*).

The Panels area is not visible by default. It is typically enabled and shown when the information needed is requested, such as when the debugger is enabled.

In the Panels area, there are four different windows (even though some extensions might add their own panels, such as GitLens): **PROBLEMS**, **OUTPUT**, **DEBUG CONSOLE**, and **TERMINAL**. Let's examine them in the following sections.

## PROBLEMS

With extension languages that have advanced editing features, such as AL, Visual Studio Code is able to identify code problems while typing. Problem lines have a specific colorization. There are three types of notifications: errors, warnings, and info. All of them can be shown in the **PROBLEMS** window. The following screenshot shows an example of the **PROBLEMS** window showing 4 errors and 3 warnings:

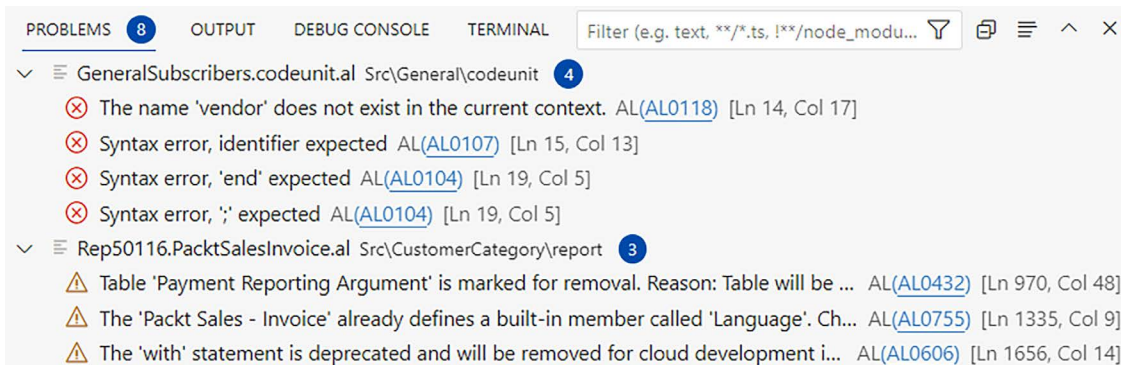


Figure 2.14: PROBLEMS window warnings

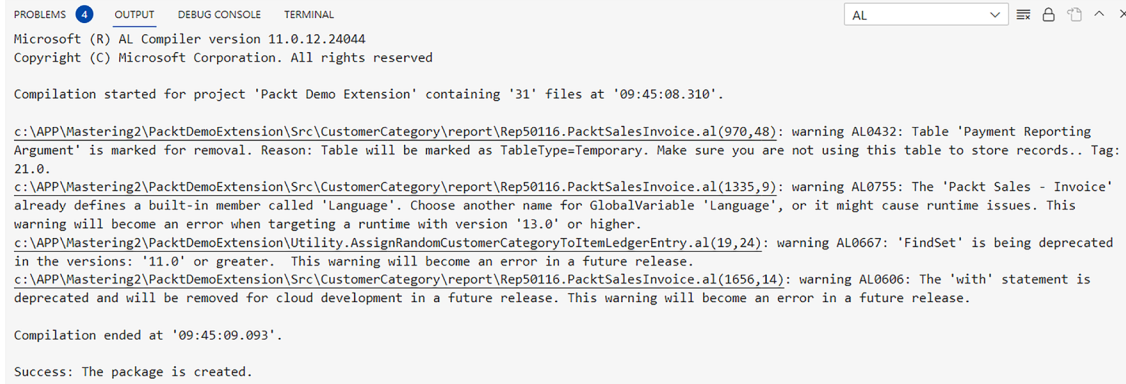
Typically, blocking errors are shown in red, while warnings are marked in yellow. Clicking any problem will open the file where the problem occurs, and the code that causes the problem will then be marked with a squiggly line in the problem color.

## OUTPUT

The **OUTPUT** panel is the place where Visual Studio Code typically displays messages during or after command execution.

Because built-in tool actions and multiple extension commands can run concurrently, it is possible to make use of a drop-down box in the **OUTPUT** panel to change the view and see the output for each standard or extension-based command.

The following screenshot shows the **OUTPUT** window in the Panels area:



```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL
Microsoft (R) AL Compiler version 11.0.12.24044
Copyright (C) Microsoft Corporation. All rights reserved

Compilation started for project 'Packt Demo Extension' containing '31' files at '09:45:08.310'.

c:\APP\Mastering2\PacktDemoExtension\Src\CustomerCategory\report\Rep50116.PacktSalesInvoice.al(970,48): warning AL0432: Table 'Payment Reporting Argument' is marked for removal. Reason: Table will be marked as TableType=Temporary. Make sure you are not using this table to store records.. Tag: 21.0.
c:\APP\Mastering2\PacktDemoExtension\Src\CustomerCategory\report\Rep50116.PacktSalesInvoice.al(1335,9): warning AL0755: The 'Packt Sales - Invoice' already defines a built-in member called 'Language'. Choose another name for GlobalVariable 'Language', or it might cause runtime issues. This warning will become an error when targeting a runtime with version '13.0' or higher.
c:\APP\Mastering2\PacktDemoExtension\Utility.AssignRandomCustomerCategoryToItemLedgerEntry.al(19,24): warning AL0667: 'FindSet' is being deprecated in the versions: '11.0' or greater. This warning will become an error in a future release.
c:\APP\Mastering2\PacktDemoExtension\Src\CustomerCategory\report\Rep50116.PacktSalesInvoice.al(1656,14): warning AL0606: The 'with' statement is deprecated and will be removed for cloud development in a future release. This warning will become an error in a future release.

Compilation ended at '09:45:09.093'.

Success: The package is created.

```

Figure 2.15: OUTPUT window messages

When working with Dynamics 365 Business Central extensions, the AL Language output window is selected automatically.

## DEBUG CONSOLE

This is a special window used by native and extension-based debuggers, such as the AL language debugger, to display information about code execution. This window and its output will be analyzed in detail in *Chapter 10, Debugging*.

## TERMINAL

Visual Studio Code allows us to execute commands in the same way as Command Prompt, directly from within the development environment. The Terminal session is based on PowerShell by default.

Now that we have all the elements that are related to Visual Studio Code in place, we can move on to the next section and deep-dive into the powerful editing features that it offers.

# Visual Studio Code – the editing features

Visual Studio Code provides many of the features that you would expect from the best-in-class code editor. If you are familiar with Visual Studio, you might have noticed that some features have been engineered in a similar way.

*Developed by developers for developers*, Visual Studio Code has keyboard shortcuts for almost every editing command, giving you the option to edit code faster and completely forget about the mouse. Let's study the most used features in the following sections.

## Comment lines

Visual Studio Code provides out-of-the-box commands for text selection and professional editing in the **Edit** menu. The **Edit** menu also includes **Toggle Line Comment**, which adds a line comment for the selected line. This means that if you select 10 lines of code, Visual Studio Code will comment out the selected lines. The beauty of this command is that it works in reverse as well. If you select the 10 commented lines and press **Toggle Line Comment**, the comments will be magically removed. It is also possible to select **Toggle Block Comment** and revert this back (*Shift + Alt + A*).

For developers working with **CSIDE**, the old legacy language for on-premises Dynamics 365 Business Central, this command is the equivalent of **Comment Selection** (*Shift + Ctrl + K*) and **Uncomment Selection** (*Shift + Ctrl + O*).

## Delimiter matching

Visual Studio Code can detect pairs of delimiters, such as brackets and parentheses. This feature is helpful if you want to delimit code blocks, and it kicks in when the mouse is placed near one of the delimiter pairs:

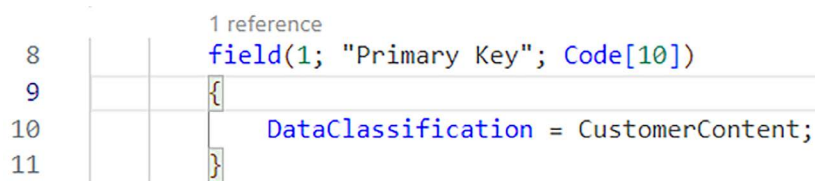


Figure 2.16: Delimiter brackets

Thanks to this feature, we can easily see that the brackets on lines 9 and 11 are matching. This is a simple example, but when working with code that uses multiple parentheses, it can be a helpful quality-of-life feature to quickly see which bracket corresponds to which.

## Text selection

The **Selection** menu also has commands that relate to text selection, but most of them are used to move or duplicate lines of code above and below the selected line.

If you position the cursor near an AL function, variable, or constant, you can use **Add Next Occurrence** (*Ctrl+ D*), **Add Previous Occurrence**, or **Select All Occurrences** (*Shift+ Ctrl+ D*) to select occurrences of the selected item, and occurrences will be highlighted in a different color.

## Code block folding

If you hover over line numbers in the code editor, a downward-pointing arrow will appear close to the initial part of a code block. Click on it to fold it, and a right arrow will appear. Click on this, and the code block unfolds:

```

1  codeunit 50103 "PKT General Event Subscribers"
2  {
3      SingleInstance = true;
4
5  > [EventSubscriber(ObjectType::Table, Database::"Item Ledger Entry",
17
18 > [EventSubscriber(ObjectType::Codeunit, Codeunit::"Release Purchase
32
33 }

```

Figure 2.17: Folded code on lines 5 and 18

The preceding screenshot shows two folded code blocks, depicted with right arrows.

## Multiple cursors (or multi-cursors)

Each cursor operates independently. *Alt* + click will generate a secondary cursor at the desired position.

The most common development situation in which you want to go for multiple cursors is when you need to add or replace the same text in different positions but within the same source file. The following screenshot shows three cursors in action when editing the AL Language **DataClassification** property:

```

6  fields
7  {
8      1 reference
9      field(1; "Primary Key"; Code[10])
10     Dataclassif
11     DataClassification DataClassification property
12     4 references
13     field(2; "Minimum Accepted Vendor Rate"; Decimal)
14     Caption = 'Minimum Accepted Vendor Rate for Purchases';
15     Dataclassif
16     3 references
17     field(3; "Gift Tolerance Qty"; Decimal)
18     Caption = 'Gift Tolerance Quantity for Sales';
19     Dataclassif
20     Dataclassif
21

```

Figure 2.18: Multi-cursors on lines 10, 15, and 20

This is a great feature for AL language developers, especially when they have to write down the same sentence many times in the same place (for example, **Caption** or **DataClassification** in a table object and for each table field).

## Mini-map

Sometimes, when working with very long files such as **report definition language (RDL)** files or codeunits, it is difficult to know where the pointer should be positioned – or is positioned – within a source file. Visual Studio Code has a fully fledged mini-map feature: a small preview of the source code file. The following is an example of an RDL:



Figure 2.19: Mini-map for an RDL file

The mini-map feature can be disabled/enabled through **View | Appearance | Minimap**, or by running the **Command Palette (F1)** and selecting **View: Toggle Minimap**.

## Breadcrumbs

The **Show Breadcrumbs** command is available by clicking **View | Appearance | Breadcrumbs** or via the **Command Palette (F1)** and selecting **View: Toggle Breadcrumbs**. With AL files, you can click each element of the breadcrumb to inspect and explore the levels in the current object. There is a bit of text in the top-left corner of the code editor that can be expanded to easily double-check the definitions of properties, functions, fields, keys, and so on. If you click on an element in the expanded list, the cursor will jump to its primary definition, making code navigation quite fast and productive.

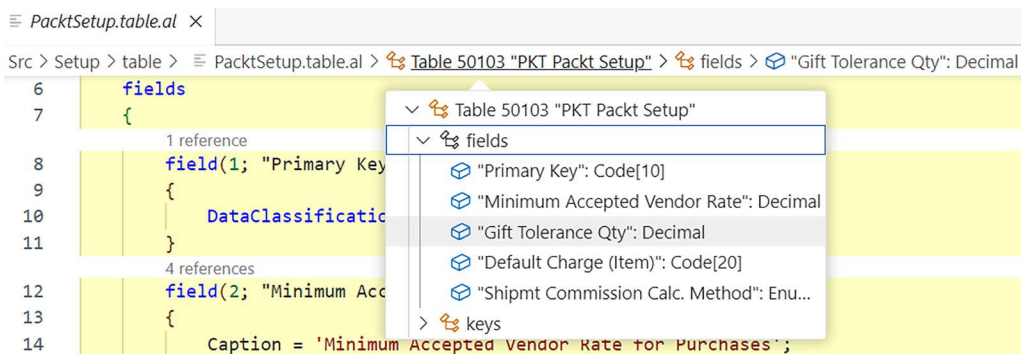


Figure 2.20: Viewing breadcrumb elements

## IntelliSense

In visual editors, **IntelliSense** is a word completion tool that appears as a pop-up list while you type. Visual Studio Code IntelliSense can provide smart suggestions, showing the definition and purpose – like online help – related to a specific element. The following screenshot shows IntelliSense in action:

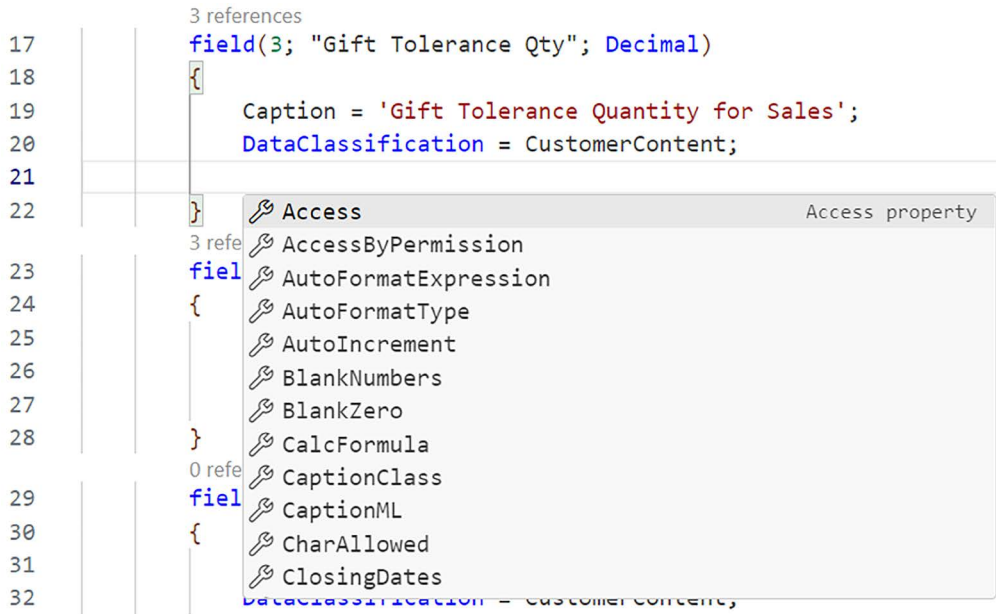


Figure 2.21: IntelliSense word completion

IntelliSense is context-sensitive, and if you need to enable it directly without typing anything, just press `Ctrl + spacebar`. Depending on the context where the cursor is placed, IntelliSense will show all the items that can be used in that context. For example, inside a Table Field declaration, it will list all the specific field properties, such as `Caption` and `CaptionML`, while in an empty codeunit definition, it will show all the properties that are exposed by a codeunit object.

## Word completion

Through the IntelliSense feature, the code editor in Visual Studio Code implements word completion for all native (such as JSON) and extension-based supported languages (such as AL). Just press `Enter` or `Tab` to insert the suggested word.

## Go to definition

This is a super-cool, must-know feature. You can hover over a variable, constant, function, or whatever code element you want with the mouse, and if you press `Ctrl`, the word or identifier (known also as a symbol) will magically switch into a hyperlink.

If you click on the word while pressing `Ctrl`, you will be automatically redirected to the code that defines that word. Pressing `Ctrl` + hovering over a code element also enables the **Go To Definition** feature.

Other possible ways to enable this feature are as follows:

- Select a code element and press *F12*.
- Right-click on a code element and then select **Go To Definition** from the context menu.

## Find all references

**Find All References** makes it very easy to parse how many times and where an object, a function, or any code element has been used across source code. You can simply right-click on any variable, function, or element name and then select **Find All References**, or use the keyboard shortcut *Shift + Alt + F12*.

When it's enabled, the code editor will create a result list in the activity bar showing how many times it has been referenced, and in which object files and position(s). A shortcut icon is created in the sidebar called **References**. The following screenshot shows how to find all references in AL files for a specific variable:

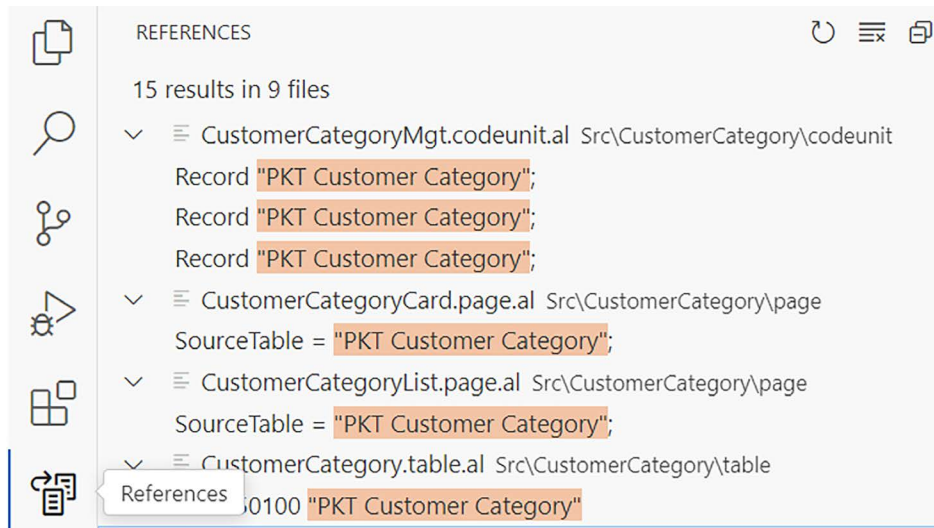


Figure 2.22: Finding multiple referenes in source code

If you expand an occurrence in the references list on the left and click on a record, the code editor will open the file where it is referenced and position the cursor in editing mode, selecting the element searched in that file.

The references list can be cleared and refreshed, and you can collapse all the elements in it. If you clear the list, you can always run the previous search again, since the history is maintained for you.

## Peek definition

Imagine that you have a large number of code files, and you need to edit the definition of a variable or field that you are currently using. With many other editors – or development environments – you most likely have to save all the files in text format, then search through all these code files and be sure to replace that variable name. This task not only can be annoying but can also distract you from the original code you were writing.

Visual Studio Code brilliantly solves this problem by providing the Peek feature, which can be enabled in different ways:

- Right-click a variable, field, or function name and select **Peek Definition**.
- Use the **Alt + F12** keyboard shortcut.

An interactive pop-up window should appear, showing the source code that defines the selected element.



Figure 2.23: Using the Peek feature to locate a definition

The above screenshot shows the **Peek Definition** for a table source, bound to a dataitem element in a report. You can then see what has been written and directly edit it.

## Renaming symbols

For a developer, it is very common to rename a variable, constant, field, or function. These coding elements are technically called symbols. Visual Studio Code provides a very powerful feature to rename symbols.

If you press **F2** on the coding element that you wish to rename, or right-click and then select **Rename Symbol**, a small interactive popup appears in edit mode. There, you can write the new element name without using a distracting dialog window, allowing you to concentrate on your coding. All references to that code element will be renamed accordingly. The following screenshot shows renaming a procedure symbol reference:

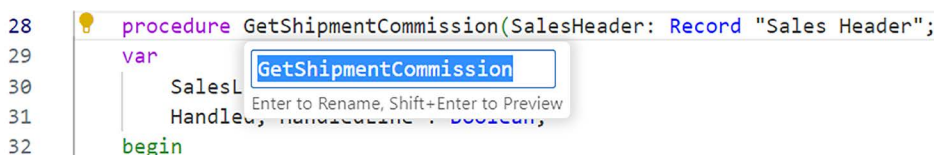


Figure 2.24: Renaming a symbol

All the features shown so far are the most useful features provided by Visual Studio Code that support proficient code editing for AL developers. At this stage, then, it is important to take a closer look at the AL Language extension and see how to configure it to achieve more from the development environment.

## Understanding the AL Language extension

AL is now a cross-platform language that is deployed through an extension for Visual Studio Code. This extension not only supports deployment on Windows OSs but is also supported for the macOS version of Visual Studio Code, and recently, it is also supported on Linux.

The free AL Language extension is available for download from the Visual Studio Code marketplace (<https://marketplace.visualstudio.com/items?itemName=ms-dynamics-smb.al>). This provides an optimized experience for Dynamics 365 Business Central extension development and includes all the support and tools that you need to build apps, including the debugger.

You might also want to install other extensions that add more languages (such as PowerShell), tools (such as Docker), or enhanced editing features to the AL Language extension. A list of the most useful marketplace extensions used by Dynamics 365 Business Central in combination with AL Language will be provided in *Chapter 17, Useful and Proficient Tools for AL Developers*.

## AL Language

Created by the Dynamics 365 Business Central modern development team, AL Language is the official Visual Studio Code extension for developing apps for small, single-tenant personalization and complex add-on vertical solutions that are deployed through the online Dynamics 365 Business Central AppSource marketplace.

The AL Language extension can be deployed in two different ways:

- Directly, as a downloadable package from the Visual Studio Code marketplace
- Manually, as an installable package (a file with a `.vsix` extension)



The manual way is only used when a specific version for a specific implementation is needed.

To start directly with AL Language, download it from the marketplace by following these simple steps:

1. Run Visual Studio Code.
2. Click on the **Extensions** view bar.
3. In the search field, type *Dynamics 365 Business Central*.
4. Select **AL Language extension for Microsoft Dynamics 365 Business Central**.

- Click on **Install**, and when the installation finishes, reload Visual Studio Code as requested. It shows the following AL Language extension:



Figure 2.25: AL Language extension

By hovering over the extension, a more verbose pop-up pane should appear with more info:

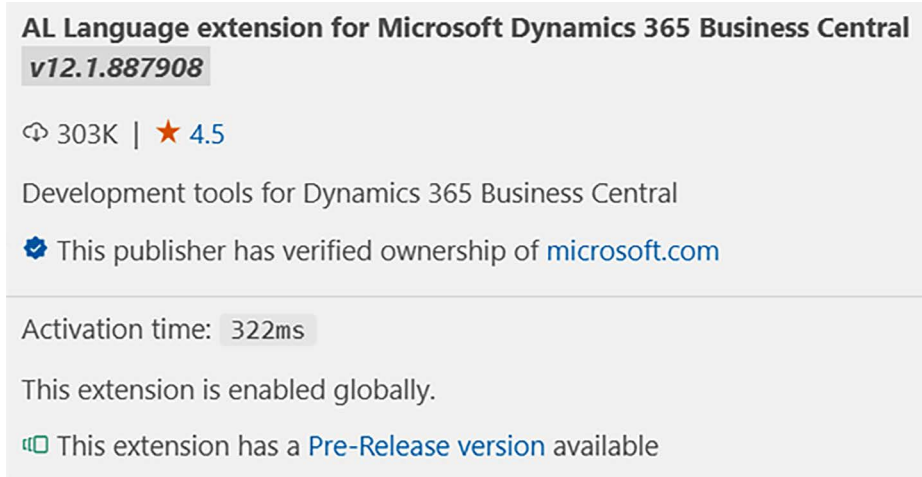
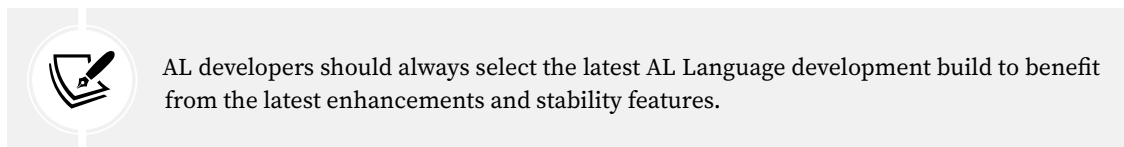


Figure 2.26: AL Language extension description

The AL Language build number, also known as the development build, is shown close to the title. In the preceding screenshot, the AL Language development build (or runtime) version is **12.1.887908**.

The development build is very important because new language features and enhancements are typically not backported to older builds, so they could be outdated and not compatible with the more recent Dynamics 365 Business Central platform updates.



Right after installing the AL Language extension from scratch, the **AL Home** page is displayed. This page contains tips for getting started with AL development and, most importantly, feeds from the Microsoft product group on what's new, announcements, or development best practices. See below for what it might look like:



## Business Central developer news

### 2023-11-16 Are you looking into creating generative AI solutions for Business Central?

Then read on here. As you move forward with exploring generative AI, we've published a couple of articles to get you started. There's a new article for the [PromptDialog page type](#) and we've also updated the reference documentation for the following properties: `PromptMode` `IsPreview` `PageType`

This serves as a starting point for other documentation that we'll be shipping on how to get started developing a generative AI solution with Business Central.

Remember that you can go to the [aka.ms/BCTech](https://aka.ms/BCTech) repo to discover more and maybe get inspired.

At Directions EMEA 2023, the keynote presentations demonstrated how Business Central has integrated AI to enhance productivity. Check these recordings out [here](#).

☒ Show at startup

## Getting started with AL development

To get started writing extensions for Dynamics 365 Business Central you will need a Business Central tenant, Visual Studio Code, and the AL Language extension for Microsoft Dynamics 365 Business Central.

[Getting Started with AL](#)

## Learn

Learn new skills and discover the power of Business Central with step-by-step guidance. Start your journey today by exploring our learning paths and modules.

[Learn resources](#)

Figure 2.27: AL Home page

Unchecking **Show at startup** will turn off popping up the AL Home page every time you start a new instance of Visual Studio Code or reload Windows. It is possible to completely turn this off or simply have it shown only when there are updates in the feed section. We will see how to configure it later in this chapter. In any case, you could always run AL Home from the Command Palette (*F1*) by using the command `AL : Home`.

The AL Language development model relates to creating, editing, and organizing flat text files with the typical `.al` extension. In short, AL Language development is simply folder- and file-based.



It's worth mentioning that the Visual Studio Code terminology calls a root folder a workspace. The AL Language root folder represents the source code container for an extension. Therefore, the AL Language root folder is also called the Visual Studio Code development workspace. Keep in mind that this is completely different from a code-workspace file.

When creating whatever kind of AL extension, the workspace consists of the following items:

- A `launch.json` file
- An `app.json` file
- Symbol files

- .al object files (such as table, page, report, and codeunit)
- Complementary files (such as the extension logo in .bmp format, translation .xlf files, and so on)

We will analyze AL Language objects and complementary files in more depth in later chapters. We will focus now on the backbone of the app development: `launch.json`, `app.json`, and symbol files.

## launch.json

This file is stored in the extension's workspace in a subfolder called `.vscode` and mainly determines the specific parameter settings – a sort of connection string – for downloading and uploading AL Language commands.

The following table shows the download and upload AL commands:

Download commands	Upload commands
AL: Download symbols	AL: Publish ( <i>F5</i> )
AL: Download source code ( <i>F7</i> )	AL: Publish and open in the designer ( <i>F6</i> )
	AL: Publish without debugging ( <i>Ctrl + F5</i> )
	AL: Rapid application publish ( <i>Alt + F5</i> )
	AL: Rapid application publish without debugging ( <i>Ctrl + Alt + F5</i> )

Table 2.1: AL commands for downloading and uploading

It is also used just to establish a connection and download symbols at will, as in the case of the **AL: Debug without publishing** (*Ctrl + Shift + F5*) command, or to launch a specific debugging feature such as, for example, **AL: Open Events Recorder**.

The `launch.json` file is a JSON object, and one of its elements is a JSON array that might have different JSON values, each representing a set of attributes targeting different deployments: on-premises or SaaS. Attributes could be mandatory or optional, depending on the target deployment.

The following table shows the `launch.json` attributes:

Attribute: Description	Deployment type
name: Shown also in the debugger window, this is used to identify the set of launch parameters. Default values: "Your own server" (on-premises), "Microsoft cloud sandbox" (SaaS).	All
type: Constant value: <code>al</code> .	All
request: Default value: "launch". For debugging purposes, it could also be "attach" (to a specific session) or "initializeSnapshot" (when initializing a snapshot debugger session). Both will be covered in detail in <i>Chapter 10, Debugging</i> .	All

Attribute: Description	Deployment type
startupObjectType: Object type to run after publishing: "Page", "Query", "Report", or "Table".	All
startupObjectId: Used together with StartupObjectType. Defines the object ID to run.	All
tenant: AAD tenant (SaaS) or tenant name (on-premises) to connect to extract symbols and/or publish the extension package.	All
environmentType: "OnPrem", "Production", or "Sandbox".	All
environmentName: Name of the environment to which to connect.	All
startupCompany: This is the company that should be accessed right after publishing.	All
breakOnError: Specifies whether the debugger should stop when an error occurs. It is also possible to exclude try functions ("ExcludeTry").	All
breakOnErrorWrite: Specifies whether the debugger should stop on record changes (insert, modify, and delete). It is also possible to exclude temporary records ("ExcludeTemporary").	All
schemaUpdateMode: Determines the data synchronization mode. Possible values are: <ul style="list-style-type: none"> <li>"Synchronize": This is the default value. If there is already data deployed for this extension, it will be preserved and not removed. The extension metadata will be synchronized with the existing one, if any.</li> <li>"Recreate": Wipes out previous metadata (tables and table extensions, typically) and uses the new metadata from scratch.</li> <li>"ForceSync": Forces the schema synchronization. This should be used with extreme caution since it might lead to data loss scenarios.</li> </ul>	All
dependencyPublishingOption: This applies in complex scenarios where multiple dependent apps are loaded from the same root folder. Possible values are: <ul style="list-style-type: none"> <li>"Default": Enables rebuilding and publishing all dependent apps.</li> <li>"Ignore": Does not apply dependency publishing. This option should be used quite carefully since it risks breaking existing interdependent apps.</li> <li>"Strict": Publishing will fail if there are any installed extensions that have a dependency on the root folder.</li> </ul>	All
server: Dynamics 365 Business Central Server URL.	On-premises
serverInstance: Dynamics 365 Business Central Server service name.	On-premises
authentication: "Windows", "UserPassword", or "AAD".	On-premises

Attribute: Description	Deployment type
primaryTenantDomain: This is mandatory only with AAD authentication for on-premises deployments. It is the URL of the AAD.	On-premises
port: Dynamics 365 Business Central development port number.	On-premises
applicationFamily: Used to develop an embedded extension for AppSource. This is a tag for Microsoft to determine the targeted upgrade operation if specific AppSource extensions have been deployed in the tenant.	AppSource
launchBrowser: Specifies whether to launch a browser when publishing extensions.	All
enableLongRunningSqlStatements: Enables the capability of displaying long-running T-SQL statements while debugging.	All
longRunningSqlStatementsThreshold: The minimum value to consider as a long-running statement for a SQL query. Default value: 500 msec.	All
numberOfSqlStatements: Sets the number of SQL statements to be shown while debugging. Default value: 10.	All
enableSqlInformationDebugger: Enables the capability of retrieving T-SQL query information.	All
disableHttpRequestTimeout: Used to avoid timeouts when debugging web service calls.	All
forceUpgrade: Forces codeunits to re-run upgrades.	All
usePublicUrlFromServer: If set to false, it will use the value specified in the server parameter instead of the one set server side as PublicWebBaseUrl in the customsettings.config file.	On-premises
useSystemSessionForDeployment: To prevent debugging, installs and upgrades codeunits (since they will run in another session).	All

Table 2.2: launch.json attributes

If you have set up more than one value in the JSON array, when you upload or download AL Language commands, you will be prompted to choose one of the parameter set names defined in the JSON array. Below is an example of a launch.json file with multiple elements:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Docker Sandbox US 23.1",
      "request": "launch",
      "type": "al",
      "environmentType": "OnPrem",
      "server": "http://BC-23-1-US",
    }
  ]
}
```

```

        "serverInstance": "BC",
        "authentication": "UserPassword",
        "startupObjectId": 22,
        "startupObjectType": "Page",
        "breakOnError": "All",
    },
    {
        "name": "PACKT Cloud Sandbox",
        "request": "launch",
        "type": "al",
        "environmentType": "Sandbox",
        "environmentName": "SandboxUS",
        "startupObjectId": 22,
        "startupObjectType": "Page",
        "launchBrowser": true,
    }
]
}

```

## app.json

Typically stored in the extension workspace root folder, `app.json` represents the extension manifest, written in JSON. Inside the JSON file, there are parameters referencing base and system application dependencies, the application parameter, the platform property, and runtime definitions. These terms need to be well understood when developing for Dynamics 365 Business Central, so we will briefly go over them next.

### Base/system application dependencies

With the October 2019 major release (version 15.x), Microsoft converted all legacy C/AL code into AL objects. Currently, the old application monolith has been split into two main extensions:

- **System application:** With approximately 1,500 objects, it is growing release after release thanks to the open-source community contribution.
- **Base application:** Depending on the localized version, it spans from 7,000 to 9,000 objects, approximately.

To be extended, these need to be referenced as dependencies in the `app.json` file, and their symbols pulled from an on-premises or an online sandbox through the *AL: Download symbols* AL Language command.

Defining dependencies to the system and base application is super simple: just set the "application" parameter with the value of the major (or major.minor) version that your extension is targeting like, for example, "application": "23.0.0.0". This will instruct Visual Studio Code, when downloading symbols, to check and request the system and base application version to have a value equal to or higher than the one specified in the parameter.

## Application parameter and platform property

When pulled in the `.alpackages` folder, symbols are typically referenced through a version number in the major, minor, build, and revision notations, and this is shown in the name of the symbols that are downloaded (for example, `Microsoft_System_Application_23.0.36560.0` and `Microsoft_Base_Application_23.0.36626.36918`).

The major version digit typically corresponds to the Dynamics 365 Business Central major update release. The 2023 Wave 2 release update is major version 23. The 2024 Wave 1 update release will be major version 24, and so on.

The minor version typically corresponds to minor updates. The November 2023 Update 1 is minor version 23.1, the December 2023 Update 2 will be minor version 23.2, and so on.

The build number is a progressive number that is incremented by Microsoft as soon as there are changes committed to the branch that are related to feature enhancements or bug fixing.

When developing an extension, you must be aware of what system and application object level is needed as a minimum requirement, as defined in the **application** parameter of the *app.json* file.

When targeting a platform for development, you must be aware of the minimum requirements held by files, AL Language statements, and APIs in order to use their features, properties, and functions. If the minimum requirements are not met, these files, statements, and APIs could be exposed to avoid unpredictable behaviors from the application. To help with this, the **platform** property represents the results of the final compilation of the Dynamics 365 Business Central platform components (client, server, web server, and so on). This property is shown with the same notation as the application.

The application parameter and platform property typically have a different value, since platform code changes and application code changes follow different compilation paths and are merged in the end, right before performing classic and regression tests.

## Runtime

Runtime represents the results of the final compilation of the Dynamics 365 Business Central AL Language extension file.

The notation is simpler and consists of a major, minor, and build version. For example, the Spring 2018 update (or the April 2018 update) is named major version 1, and it increases every year. The current runtime version that targets the 2023 Wave 2 platform and application has version number 12. It is worth noting that the runtime version does not correspond with the application version, so it is important to use the right combination of application and runtime together.

When developing extensions, within the *app.json* file, you can define what runtime version the application is targeting. This enables or disables different sets of features that can or cannot be part of the target platform deployment, and the AL Language extension's runtime will detect that. For example, by specifying `"runtime": "12.0"`, we are targeting all modern development features included with the 2023 Wave 2 release and upward. The following table shows the main *app.json* attributes:

Attribute	Description
Id	<b>Global Unique Identifier (GUID)</b> of the extension.
Name	Extension name.
publisher	Publisher name.
version	Version of the extension package with major, minor, build, and revision.
Brief	Short description of the extension.
description	Long and verbose description of the extension.
privacyStatement	URL to the privacy statement.
EULA	URL to the license terms and conditions for the app.
Help	URL to app helpdesk support.
url	URL to the extension package's home page.
Logo	Relative or full path to the app logo from the root directory of the extension.
dependencies	List of dependencies from other extensions.
screenshots	Relative or absolute path to app screenshots.
platform	Minimum platform version supported.
idRanges	Range of application object IDs or an array of object ID ranges.
resourceExposurePolicy	It substitutes the now obsolete <code>showMyCode</code> parameter and is used to granularly set the accessibility of the source code and symbols. It will be discussed later in this chapter.
Target	Default value: Cloud. Set this value to OnPrem if you need to target the extension to on-premises deployment. Be well aware that the Universal Code initiative will apply higher fees year over year if you choose to target on-premises deployments.
helpBaseUrl	URL for the extension's online help.
contextSensitiveHelpUrl	URL for the context-sensitive help for an AppSource extension.
supportedLocales	Comma-separated list of the local languages supported by the extension.
features	Optional features that could be enabled by the compiler. It will be discussed later in this chapter. An example is <code>TranslationFile</code> .
runtime	Minimum runtime version targeted by the extension.
source	Optional. It might contain source DevOps information such as the URL address of the source repo and the Git commit ID that triggered the extension build.
build	Optional. Build DevOps info such as the agent's name and the URL at which to find the resulting build.

Table 2.3: *app.json* attributes

The following is an example of a condensed `app.json` file:

```
{
  "id": "dd03d28e-4dfe-48d9-9520-c875595362b6",
  "name": "Packt Demo Extension",
  "publisher": "PACKT",
  "brief": "Customer Category, Gift Campaigns and Vendor Quality Management",
  "description": "Customer Category, Gift Campaigns and Vendor Quality
Management",
  "version": "1.0.0.0",
  "privacyStatement": "",
  "EULA": "",
  "help": "",
  "url": "http://www.demiliani.com",
  "logo": "./Logo/ExtLogo.png",
  "dependencies": [],
  "screenshots": [],
  "platform": "1.0.0.0",
  "application": "23.0.0.0",
  "features": [
    "NoImplicitWith", "TranslationFile"
  ],
  "idRanges": [
    {
      "from": 50100,
      "to": 50149
    }
  ],
  "contextSensitiveHelpUrl": "https://PacktDemoExtension.com/help/",
  "resourceExposurePolicy": {
    "allowDebugging": true,
    "allowDownloadingSource": true,
    "includeSourceInSymbolFile": true,
    "applyToDevExtension": false
  },
  "runtime": "12.0"
}
```

Two of the main `app.json` parameters need a deeper analysis: **features** and `resourceExposurePolicy`.

The features parameter currently admits specifying the following placeholders:

- **TranslationFile:** Adding this parameter flag in features enables the generation of a directory called **Translations** in the extension folder and an **.xlf** translation file containing all the labels used in all extension objects. Translation files will be handled in deep detail in *Chapter 4, Developing a Customized Solution for Dynamics 365 Business Central*.
- **GenerateCaption:** If **TranslationFile** is set, it will automatically generate translation placeholders for all objects that do not have a **Caption** or **CaptionML** property explicitly specified.
- **GenerateLockedTranslations:** This will generate the appropriate elements for locked labels in the translation file when building the extension.
- **AllTranslationItems:** This will generate translation placeholders for all possible object elements in the extension.
- **ExcludeGeneratedTranslations:** This excludes the generated translation files from the extension.
- **NoImplicitWith:** By adding this flag, you instruct the compiler to switch off the use of the **With AL** statement and all implicit field definitions (e.g., within pages, you should always reference **Rec.<fieldname>**).
- **NoPromotedActionProperties:** This will not admit any old promoted property definition in the extension and will switch on the new modern action reference syntax.
- **UseLegacyAnalyzerStrategy:** This reverts to the previous – and quite expansive in terms of resources – code analysis where every change in the code will trigger code analysis within the entire project.

The **resourceExposurePolicy** parameter is an array of values and it is very important in defining if, when, and how to surface the code written in the extension. Currently, it supports the following Boolean elements:

- **allowDebugging:** Capability of debugging the code. It will download locally the source code of the files needed to debug in **.dal** (non-editable) format.
- **allowDownloadingSource:** Capability of switching on/off the download source button on the Extension Management page for that specific extension.
- **includeSourceInSymbolFile:** Determines whether the source code should be included or not in the symbols package when publishing the extension through PowerShell cmdlets.
- **applyToDevExtension:** By default, extensions published directly through Visual Studio Code (so-called dev extensions) always include the source code in the symbols package. Turning this flag on will perform the same action specified for **includeSourceInSymbolFile** and also for extensions that are going to be published with dev scope. It is worth noting that this will enable the ability to debug apps that have **allowDownloadingSource** turned off, if the app is deployed to a sandbox using Visual Studio Code.

This is not over. The *app.json* file is enriched with new parameters, release after release. Some of them could be considered of minor importance, considering a per-tenant extension development, but they could turn out to be vital for the clean development of ISV or AppSource extensions:

- **internalsVisibleTo**: This is an array of apps that have been granted access to the objects that are defined with `Access = Internal` in the current extension.
- **propagateDependencies**: This is used typically in large projects with a discrete dependency tree. If extension 1 depends on extension 2 and that depends on extension 3, by default, the primary extension will not be able to declare anything defined in extension 3. If extension 2 has this parameter set to `true`, then extension 1 will be able to see and declare methods defined in extension 3, without the need to explicitly add the extension manifest in the dependencies parameter of its *app.json* file.
- **applicationInsightsConnectionString**: This is used to send telemetry logs in a specific ingestion endpoint. Please note that there is no need to specify any connection string in the Dynamics 365 Business Central **Tenant Admin Center** (TAC) to have these sent. It is used with AppSource apps by ISVs that could gather information from different deployment sources to improve the extension experience or for troubleshooting purposes.
- **keyVaultUrls**: Within an AppSource app, it is typical to keep all the resource exposure policies as `false` and override them by specifying an Azure key vault that stores the AAD tenant ID(s) where some of these policies are different. Setting all resource exposure policies to `false` and implementing the override strategy is the highest level of private IP protection that could be reached for AppSource extensions.



To learn more about Azure Key Vault, please read *Dynamics 365 Business Central: Using Azure Key Vault for your secrets*: <https://demiliani.com/2020/10/06/dynamics-365-business-central-using-azure-key-vault-for-your-secrets/>.

- **suppressWarnings**: Code analyzers, which we will learn about later in this chapter, obey several error and warning rules identified by a rule ID. This parameter is a collection of warning rule IDs that should not be shown (skipped) in the Visual Studio Code **Output** window.
- **preprocessorSymbols**: This is a list of names (symbols) used by preprocessor directives. Pre-processor elements or directives are used to set regions of code that could be expanded or collapsed, to suppress warnings in a more capillary way than `suppressWarnings` does and, overall, to instruct the compiler to include or exclude specified code blocks based on the existence of specific symbols. This is a very useful technique when working with obsolete objects or fields.



To learn more about obsoleting code, please read *Best Practices for Deprecation of AL Code*: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-deprecation-guidelines>.

With this information, you should now be able to master both the `launch.json` and `app.json` extension configuration files and tweak them according to the runtime version. In the next section, we will introduce symbols and explain their vital importance in extension development.

## Understanding symbols

Like in many other languages, symbols represent references to a collection of standard objects, properties, and functions. They are a special extension file themselves with the typical `.app` naming convention and are used to maintain object reference consistency while compiling, and they also populate valid IntelliSense entries.

With Dynamics 365 Business Central, symbols are already loaded inside the application database, and these can be grouped into two classes:

- **System symbols:** Symbols for system tables in the 2000000004 to 2000000199 ID range, and also virtual table definitions, plus some system codeunits. All these structures cannot be modified through extensions. Typically, these are downloaded from any environment as a `System.app` file and rarely are changed within a major version.
- **Extension symbols:** All other symbols are typically generated when compiling (*Ctrl + Shift + B*) any extension and are part of the app package.

Whenever you extend an application, you always need to have the appropriate symbols downloaded and in place. You can achieve this in two ways:

- Connect to an online sandbox environment, run the Command Palette (*F1*), and type and select **AL: Download Symbols**.
- Copy the required symbols manually from another place (such as the product DVD, for on-premises deployment) and store them in the defined symbol storage directory.

If you have a multiuser environment with developers that are working on the same staging tenant, you might think of downloading symbols through the Command Palette once and then setting a common path for storing the symbols for all users. In this way, it is possible to avoid downloading the same set of symbols every time, thereby increasing development productivity.

The default symbol-storing path can be changed using one of the following shortcuts:

- From the **Menu** bar, go to **File** (*Alt + F*) | **Preferences** (*P*) | **Settings** (*S*), and then select **AL Language extension** settings.
- Use the settings shortcuts (*Ctrl + ,*) and then select **AL Language extension** settings.

The parameter to change is **Package Cache Path**, the default value of which is set to the relative path, `./alpackages`.

Alternatively, you could run the Command Palette (*F1*), type and select **Preferences: Configure language-specific settings...**, then choose **AL**. The `settings.json` file will open, and you can then add or change the values of the `al.packageCachePath` parameter. Later in this chapter, we will also discuss other AL Language configuration settings.

Together with the system application extension, base application extension, and system symbols, your extension might also depend on other custom or third-party extensions. These extensions, then, should emit symbols that you should be able to download from the application database when invoking **AL: Download Symbols** from the Command Palette.

To specify that your extension has a dependency on another extension(s), you must populate the dependencies parameter in the `app.json` file. This is what the `app.json` file parameter looks like for an extension that depends on another app:

```
"dependencies": [
  {
    "id": "dd03d28e-4dfe-48d9-9520-c875595362b6",
    "name": "Packt Demo Extension",
    "publisher": "PACKT",
    "version": "1.0.0.0"
  }
],
```



If you have installed waldo's CRS AL Language Extension (<https://marketplace.visualstudio.com/items?itemName=waldo.crs-al-language-extension>), you could type `tdependencywaldo` to enable the code snippet to easily edit each JSON array element for this parameter. This will make your coding faster and prevent syntax errors. We will discuss the standard and custom code snippet features in the last section of this chapter and more on productive extensions in *Chapter 17, Useful Visual Studio Code Extensions for Dynamics 365 Business Central Developers*.

The version parameter of the dependent extension(s) represents the lower bound for the compiler to accept the symbols. In other words, symbol versions of the dependent extension lower than the one reported are not considered valid for download or compile operations.

## Inside symbols

Symbols are the result of a compression operation of several files that are used by the AL Language extension. To demonstrate what is under the hood, use a decompression tool (for example, 7-Zip) to extract their content after renaming the `.app` package with the `.zip` extension.

To give a practical example, the following tables show the standard symbol components (files and directories) for a standard base application extension:

Filename	Description
[Content_Types.xml]	Specifies the content of the package. For example: <code>.xml</code> , <code>AL</code> , <code>.json</code> , <code>.rdlc</code> files, and so on.
MediaIdListing.xml	Specifies the extension logo filename and its ID.
navigation.xml	Contains entries for the Role Explorer.

NavxManifest.xml	Reports the manifest for the standard symbol or extension. In other words, it is a translation into XML of the app.json file.
DocComments.xml	<p>Contains all documentation comments for tables, fields, functions, and so on. Below is a simple snippet for an AL function:</p> <pre>&lt;member name="M:..Line With Price. GetPriceType:Enum::Price Type"&gt;   &lt;summary&gt;     Returns the price type that was set by the     SetLine() method:   &lt;/summary&gt;   &lt;returns&gt;The price type&lt;/returns&gt; &lt;/member&gt;</pre>
SymbolReference.json	<p>Contains all references in JSON notation to AL objects. It could also be quite a big JSON file (for example, in a standard base application, it is 60+ MB). These JSON files are heavily used by the AL Language extension to maintain reference integrity while compiling/building the app package and to enable all IntelliSense-related features. Basically, it is structured as an array containing a list of valid AL object parameters, as shown in the following snippet:</p> <pre>"Tables": [], "Codeunits": [], "Pages": [], "PageExtensions": [],</pre> <p>For each of these object elements, there are specified fields, properties, functions, and so on.</p>

Table 2.4: Symbol components



Symbol JSON files cannot be hacked/changed to manually generate or modify a symbol file.

Next, let's also have a look at what the various symbols directories do:

Directory name	Content description
entitlement	Contains an XML file that stores entitlement entries for every object.
layout	Report layouts.
logo	Extension logo.
ProfileSymbolReferences	Symbols for profiles and related page customizations.
Src	AL files. Their content is used typically to show the code while debugging.
Translations	Translation files in the XLF format.

Table 2.5: Symbol directories

Symbols are the beating heart of the extension validation mechanism and, as shown in the previous tables, they also could carry out the code, depending on how resource exposure policies have been set up in the `app.json` file.

To deeply analyze AL symbols, the Microsoft AL Language extension provides a useful tool with a few functionalities to analyze symbols, **AL Explorer** (*Ctrl + Shift + F12*):

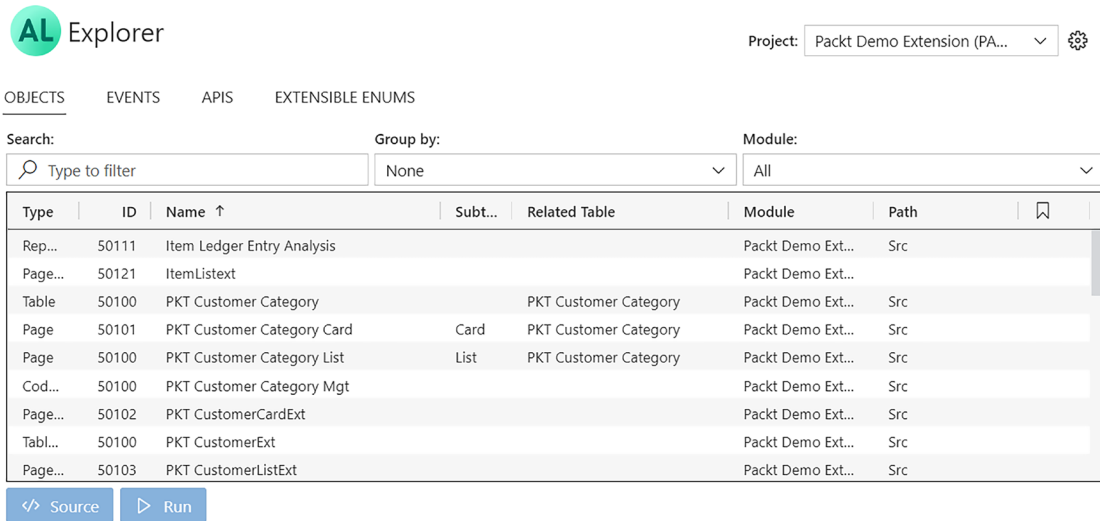
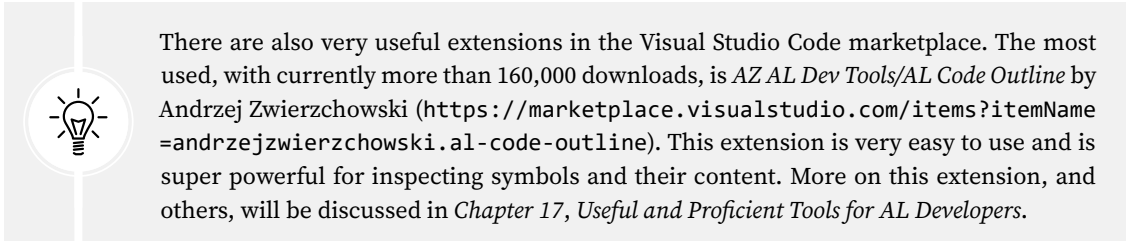






Figure 2.28: AL Explorer

This tool is currently divided into four self-explanatory tabs: **OBJECTS**, **EVENTS**, **APIS**, and **EXTENSIBLE ENUMS**. For each tab, a list of entities taken by symbols is provided and it is possible to search for names, IDs, and so on, group them by specific properties or namespaces, filter them by modules or sources, or simply choose the ones bookmarked.



Once you have identified the entity that you were looking for, you could even perform some actions. For almost all of the entities listed in the tab, you could always click on the **Source** button and inspect the code, while for runnable objects you could select the entry and click the **Run** button. Within events, you could select the publisher that you were looking for and easily click on the **Subscribe** button, to automatically add the prototype of a subscriber to your clipboard:

 Copied to clipboard. Paste this code in your application     
to create a subscriber for this event.

Source: AL Language extension for Microsoft Dynamics 365 Business Central (Ext...

*Figure 2.29: Copying a subscriber prototype to your clipboard*

You could then paste the content of the clipboard in your custom Codeunit object, without the need to type all the signature elements.

As with the **AL Home** page, it is possible to completely turn this page off so it doesn't pop up and we will see how to configure it later in this chapter. In any case, you could always run the **AL Explorer** page from the Command Palette (*F1*) by using the command *AL: Explorer* or by using the *Ctrl + Shift + F12* shortcut.

After learning about symbols and the tools to analyze them, we have completed an overview of the main items that are needed to build an app. Let us have a look now at AL Language extension configuration, and how to set it up to have a more productive development environment.

## AL Language extension settings

Per-user and per-workspace settings can be easily shown through the shortcut `Ctrl + ,` (comma). An intuitive menu will be displayed, and by selecting **Extension | AL Language extension configuration**, a set of configuration parameters is listed. The following screenshot shows the AL Language extension configuration parameters:

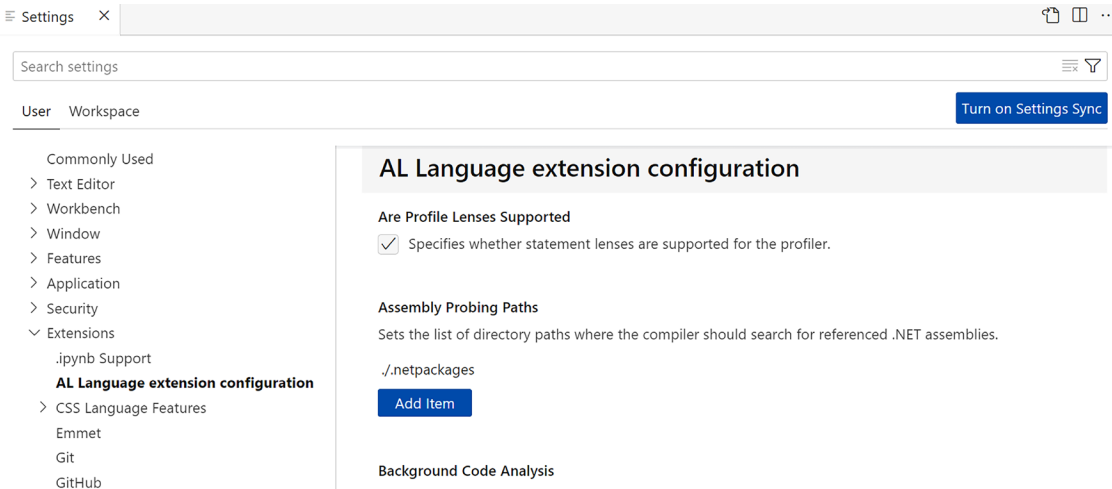


Figure 2.30: Extension configuration parameters

Basically, these configuration values are saved into a file called `settings.json`. The following lists describe the most common ones, ordered per parameter type.

Path parameters:

- `al.packageCachePath`: It is possible to change the default value to a local folder or a shared folder for multi-developer environments. This represents the path where to store and look for symbols.
- `al.assemblyProbingPaths`: This parameter is fundamental to compiling extensions when there are references to external assemblies. Its data type is a JSON array, so the developer has to specify a comma-separated list of paths where the assemblies should be stored at design time.
- `al.editorServicesPath`: If service logging is enabled, it determines where to store the service and debugger logs.
- `al.algoSuggestedFolder`: This defines where any new project folder should be created. It overrides the default creation path.

Compilation parameters:

- `al.compilationOptions`: Used to specify whether a report layout should be generated or not when compiling, if it does not exist, whether to have a serialized or parallel build of the package, and finally, the polling to emit diagnostics. Within the compilation options, it is also possible to specify the folder where the compiler should place the resulting extension file (`.app`).

### Browser parameters:

- `al.browser`: Choose your preferred browser to launch your Dynamics 365 Business Central application from Visual Studio Code or leave it to use the system default. It is useful if you have multiple browsers installed.
- `al.incognito`: Choose to start the browser in a normal session that stores existing credentials or use the private/incognito browsing mode.

### Miscellaneous parameters:

- `al.enableCodeActions`: Enables or disables code actions such as automatically converting multiple `if` statements to a `CASE` statement or kicking in spell check. By default, it is enabled. Code actions, as you might imagine, are quite expensive in terms of resource consumption.
- `al.enableScriptIntelliSense`: Enables or disables the IntelliSense for JavaScript control add-in script files.
- `al.showHomeAtStartup`: You could choose to let the **AL Home** page pop up Always, Never, or, preferably, WhenUpdated.
- `al.showExplorerAtStartup`: You could choose to let the **AL Explorer** page pop up Always, Once, or Never.
- `al.inlayhints.functionReturnTypes.enabled`: Inlay hints are additional information provided for the source code that are displayed inline. Inlay hints should be enabled by default, but you could set them to have a different behavior by changing the `parameter.editor.inlayHints.enabled` from `on` to `offUnlessPressed` or `onUnlessPressed` to have the information appear selectively, depending on whether the `Ctrl + Alt` keys are being pressed.
- `al.inlayhints.parameterNames.enabled`: As above, but info is provided for every single parameter name in the signature.

By pressing `Ctrl + Shift` in any place inside the `settings.json` file and typing `al`, you might have noticed that it will show a few other AL Language configuration parameters. These are related to code analyzers, rule sets, log paths, performance profilers, and other features that will be addressed later on or in other chapters.



If you would like to learn more about every single parameter and how to streamline your configuration for the best performance, we recommend reading *Optimize Visual Studio Code for AL Development*: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-optimize-visual-studio-code>.

After exploring the core settings that are needed to develop an extension, let us have a quick look into a notable feature offered by Visual Studio Code and the AL Language extension to better develop clean solutions according to best practices and guidelines: code analyzers.

## Understanding code analyzers

The AL Language design-time experience is greatly enhanced by code analyzers. Code analyzers are part of the standard AL Language extension and are a set of contextual rules that are applied to extension development. These rules can generate an error, a warning, or info when you're developing an extension. Since they patrol every single line or sentence of your code, they are also known as *code cops*.

Code analyzers can be enabled and disabled at will, both per workspace and globally.

To enable code analyzers, perform the following steps:

1. Go to **File | Preferences | Settings (Workspace settings) | Extension | AL language extension** and choose to edit the `settings.json` file.



You could also choose to edit the `settings.json` file through the user settings. However, since you might develop per-tenant extensions and also AppSource apps in the same environment, it would make more sense to have these enabled per workspace instead of per user.

2. In the `settings.json` file, it is possible to add or change the following relevant parameters:
  - `al.enableCodeAnalysis`: Changing this parameter to `true` enables the analyzers that are specified in the JSON array parameter `al.codeAnalyzers`.
  - `al.codeAnalyzers`: Currently, the supported values are as follows:
    - `${AppSourceCop}`: This must be enabled when developing extensions targeted for the AppSource marketplace.
    - `${CodeCop}`: This strengthens the standard AL Language development guidelines, and it is recommended to be enabled for every kind of target.
    - `${PerTenantExtensionCop}`: Together with `${CodeCop}`, this should be enabled on every development target, except when developing extensions for the AppSource marketplace, where `${AppSourceCop}` should be used.
    - `${UICop}`: This is the last addition to the code analyzers, and it checks that the code matches the features that are supported by modern clients and avoids hitting user interface limitations. It should always be enabled, like `${CodeCop}`.
  - `al.backgroundCodeAnalysis`: This defines if the analysis should also happen in the background during coding and the frequency at which it should kick in. Since code analysis is quite resource-consuming, if it kicks in too frequently, it could interfere with normal programming activity. Therefore, it is recommended to have this set per File for small to mid-sized extensions, while with quite complex solutions, it is recommended to have this changed to a more resource-savvy Project scope.

- `al.outputAnalyzerStatistics`: This parameter will generate statistics at every compilation build. It is useful to determine code cops' performance.
  - `al.ruleSetPath`: This is the path to a file that contains changes to the rules that are provided through standard code analyzers. A ruleset file is written in JSON notation and has a reference to an existing ruleset item ID that is implemented in the standard AL Language extension. This file is typically edited to redefine the importance of the rules within a specific extension project or workspace.
  - `al.enableExternalRuleSets`: This enables or disables the capability of using a URL as a path for ruleset files.
3. If we implement code analyzers in any extension project that we have created, even the super simple HelloWorld sample created through the *AL: GO!* command, it will help us to find out more info about the code style, and whether there are improvements to be applied.

Let's see what the analyzers will find out in the default HelloWorld project by changing the `settings.json` file in the workspace settings as follows:

```
"al.enableCodeAnalysis": true,
"al.backgroundCodeAnalysis": "File",
"al.codeAnalyzers": [ "${CodeCop}",
  "${PerTenantExtensionCop}",
  "${UICop}" ],
```

4. In the **PROBLEMS** window, there might now be something displayed. In this very simple case, there should be one warning, as per the following screenshot:

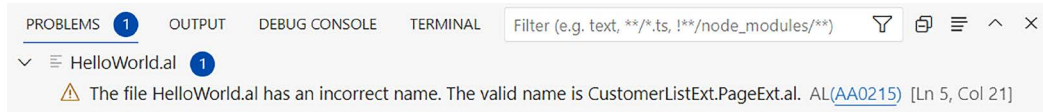


Figure 2.31: Viewing a displayed warning

Looking at the error, it is clear that the `HelloWorld.al` file does not reflect the standard naming convention and should be changed to `CustomerListExt.PageExt.al`.

It is trivial to say that if you rename the file according to what is reported by the warning, the warning will magically disappear. But what if, for some reason, we do not want to have that warning or info message displayed at all (to avoid the **PROBLEMS** window cluttering, for example)?

A rule's importance value can be changed at will or the rule itself can be suppressed by simply creating a JSON file that contains the IDs of the rules that need to be changed and how they have to be set according to your company's development rules:

1. Let's create a directory in the extension's main folder called `.ruleset`, and create a file called `demo.ruleset.json`:

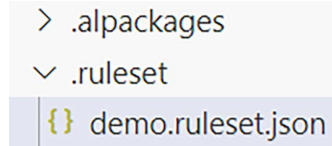


Figure 2.32: Creating a file within a folder

2. Open `demo.ruleset.json`, and invoke the `ruleset` standard snippet to write the following:

```
{
  "name": "PacktDemoExtensionRuleSet",
  "description": "Demo Rule Set for Hello World (PTE)",
  "rules": [
    {
      "id": "AA0215",
      "action": "Hidden",
      "justification": "File naming warning is kept hidden"
    }
  ]
}
```

In this way, we would like to instruct the AL Language code analyzer to avoid adding a warning record in the **PROBLEMS** window for the rule whose ID is AA0215.

3. The last step to make it work is to assign the `alRuleSetPath` parameter to point to the newly created file in the `settings.json` file:

```
"al.ruleSetPath": "./.ruleset/demo.ruleset.json"
```



When you assign the path to a ruleset file, it is recommended that you save all files and close and reopen Visual Studio Code or use the **Developer: Reload Window** command from the Command Palette, to be sure that there are no permission errors, and access the ruleset file by the current process.

Once the ruleset file is in place, there should not be any warnings in the **PROBLEMS** window related to the file. Takeaways at this stage are that developers should make good use of these rules in their own company and discuss what needs to be changed, maintained as is, or completely turned off.



Be careful when enabling code analyzers, since they might increase the memory consumption footprint in the development machine. It is recommended to turn on statistics with `"al.outputAnalyzerStatistics": true` in the `settings.json` file, if any performance problem arises, during compilation.

Now that we have mastered Visual Studio Code's main elements and features and are close to our first HelloWorld.al sample or our super sexy solution being developed, it is time to change gears and introduce GitHub Copilot to help us code faster (and sometimes better).

## GitHub Copilot for AL developers

In your daily practice, you have probably desired, at some point in time, to have someone other than the rubber duck on your desk to help you find out the right procedure or code snippet faster to finalize your code. GitHub Copilot could be your dream come true in assisting you while developing.

GitHub Copilot's documentation and price plans (currently \$10/month for individuals, \$100 per year, and \$19 per user per month for the Business edition) can be found at *this web page*: <https://github.com/features/copilot> and is available as a Visual Studio Code extension in the marketplace: <https://marketplace.visualstudio.com/items?itemName=GitHub.copilot>.



Figure 2.33: GitHub Copilot



Before installing GitHub Copilot, it is required to have a GitHub account. If you haven't already got your own, you could easily create one at <https://github.com/>. This operation is free of charge.

Let's enable the free trial together, step by step.

1. Go to <https://github.com/features/copilot> and click on **Start my free trial**.  
**NOTE:** If you have not already signed in to your GitHub account, you will be prompted to do so.
2. Give consent to the 30-day trial period.
3. Add your payment details.  
**NOTE:** Remember to confirm or remove the payment details before the end of the trial period
4. When prompted with **GitHub Copilot can allow or block suggestions matching public code**, choose **Allow**.



When you apply for a trial version, it might take up to 30 minutes to process and synchronize your submission. You should wait for this time before moving forward with the next bullet point.

5. Launch Visual Studio Code and go to **Extensions** (*Ctrl + Shift + X*)
6. Search for GitHub Copilot and install it.
7. Reload the Visual Studio Code window. You will be prompted to sign in (see below):

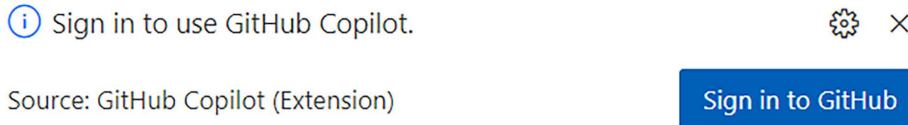


Figure 2.34: Signing in to GitHub Copilot

8. Allow and grant all prompt requests to generate the appropriate token to interact and work with Copilot.

And that's all. You should be ready to work together with Copilot right away. To be sure that Copilot is enabled, just check the small Copilot icon at the bottom right in the status bar:



Figure 2.35: GitHub Copilot icon

You can click on it if you want to disable it and go back to flying solo.

To test the effectiveness of Copilot's knowledge and how this could be of any help, just perform these simple steps in a simple `HelloWorld.al` file.

Under the `trigger` declaration, add a `var` section, then press *Enter* and *Tab*. GitHub Copilot will suggest creating a record variable called `Customer` of type `Customer`, as follows:

```
trigger OnOpenPage();
var
| Customer: Record Customer;
begin
| Message('App published: Hello world');
end;
```

Figure 2.36: GitHub Copilot suggestions

You might accept the suggestion by simply pressing *Tab* twice.

From now on, you can continue experimenting with how good, fast, and proficient it is to develop together with GitHub Copilot.

If you want to learn more about GitHub Copilot, it is worth watching *20221121 - GitHub Copilot Write Extensions for Business Central with AL*: <https://www.youtube.com/watch?v=i-K3J6C5zkc>.

## Summary

Visual Studio Code is a code-centric tool that supports, out of the box, a wide variety of languages, providing coding features such as syntax colorization, delimiter matching, code block folding, multiple cursors, IntelliSense, and so much more.

By installing the AL Language extension, this modern development environment is fully set up as an app playground for beginner and skilled developers. We have unveiled some tips and tricks in this chapter that enable you to be proficient in the developer's daily work of creating modern apps for Dynamics 365 Business Central.

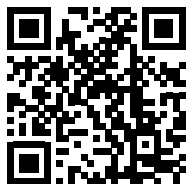
We then moved on to learn about the powerful coding features that this modern development environment offers, and, in the end, we had some fun working with GitHub Copilot and tasted a bit of how artificial intelligence could help developers.

After all of this, it is time to see AL Language in action throughout this book and move into a structured basic and advanced extension development. This is what we will do in the next couple of chapters.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>





# 3

## Extension Development Fundamentals

In this chapter, we will recap app object fundamentals. This is meant as a refresher for skilled developers, while newbies will learn about some vital building blocks. We will do an in-detail examination of the objects that are part of the extension-based development model and learn how to create new objects using a programming language known as **ApplicationLanguage (AL)**. We will then learn how to extend standard objects and how best to handle an extension project.

More specifically, we will cover the following topics:

- The basics of extension development
- An overview of the main AL objects
- How to create basic objects in an extension project
- Best practices for handling your AL projects
- Guidelines for AL objects

By the end of this chapter, you will have learned about the different AL object types, as well as how to code and use them, and, roughly speaking, you will be ready to start a Dynamics 365 Business Central extension project from scratch using AL and a modern development environment (Visual Studio Code).

### Technical requirements

To follow along with this chapter and experiment with basic object creation in AL Language, you will need the following:

- A Microsoft Dynamics 365 Business Central online or on-premises sandbox environment
- Visual Studio Code
- The AL Language extension for Visual Studio Code, installed from the Visual Studio Marketplace

## Basic concepts regarding extensions

With Microsoft Dynamics 365 Business Central SaaS, partners and customers do not have access to the database backend or the standard code base deployment.

These are the biggest differences compared with on-premises versions, where you can still have access to the Azure SQL or SQL Server database and decide if and how to deploy the standard code base. In the SaaS world, you cannot alter the database schema or the standard business logic.

Keep in mind that modifying the core is your own responsibility; Microsoft strongly discourages any direct changes to the database as well as the creation of customized monolithic base applications.

In the previous versions of Dynamics 365 Business Central, formerly known as Microsoft Dynamics NAV, and with the previous language, formerly known as C/AL, we always talked about *code modifications* or *customizations*. With the advent of the SaaS world, we must switch our developer mindset to consider a new concept: **code extensions**.

To change Dynamics 365 Business Central's application behavior, we must create extensions that add value to the current development. Therefore, we no longer customize the ERP, but we literally *extend Dynamics 365 Business Central*.

An extension (according to Microsoft's official documentation) is defined as follows:



*“An installable feature built in a way that it does not directly alter source resources and that is distributed as a preconfigured package.”*

An extension interacts with the standard code base or other first- or third-party extensions by using **events**.

An event is essentially a function that is triggered by code when something happens in a business process. This function is normally defined as the **event publisher** function. It comprises only a signature and does not execute any code. The object that contains the event publisher function is defined as the **publisher**.

In Dynamics 365 Business Central, events are classified according to the following types:

- **Database events:** These are automatically raised by the system during database operations on a table object (such as insert, modify, delete, and rename).
- **Page events:** These are automatically raised by the system during operations in a page object.
- **Business events:** These are custom events that are raised by C/AL code. A business event defines a formal contract with an implicit promise not to change in future application releases.

- **Integration events:** These are custom events that are raised by C/AL code. They are like business events, but they can change their signature in future releases of the application.
- **Global events:** These are system events that are raised by the application.

When an event is published and raised by code, it is available in the application for subscriptions. A **subscriber** is a code function that listens for and handles a published event. It subscribes to a specific event publisher function and handles the event by adding custom business logic to it. When the application raises an event, the subscriber function is automatically called, and its code is executed.

Events guarantee that you can interact with or modify the behavior of business processes without changing the first- or third-party code base.



Remember that you can have multiple subscribers to a single event publisher function. In this case, the order of the subscriber's execution cannot be determined (it is declared officially as random and can change at execution time), so be careful regarding the event chain when you architect your code. Do not stick with any kind of event order execution that has multiple subscribers for the same publisher.

Despite what is officially declared, there is an interesting blog post related to subscriber order called *Dynamics 365 Business Central: Execution order of subscriptions for the same event in different extensions (apps)*: <https://yzhums.com/22715/>.

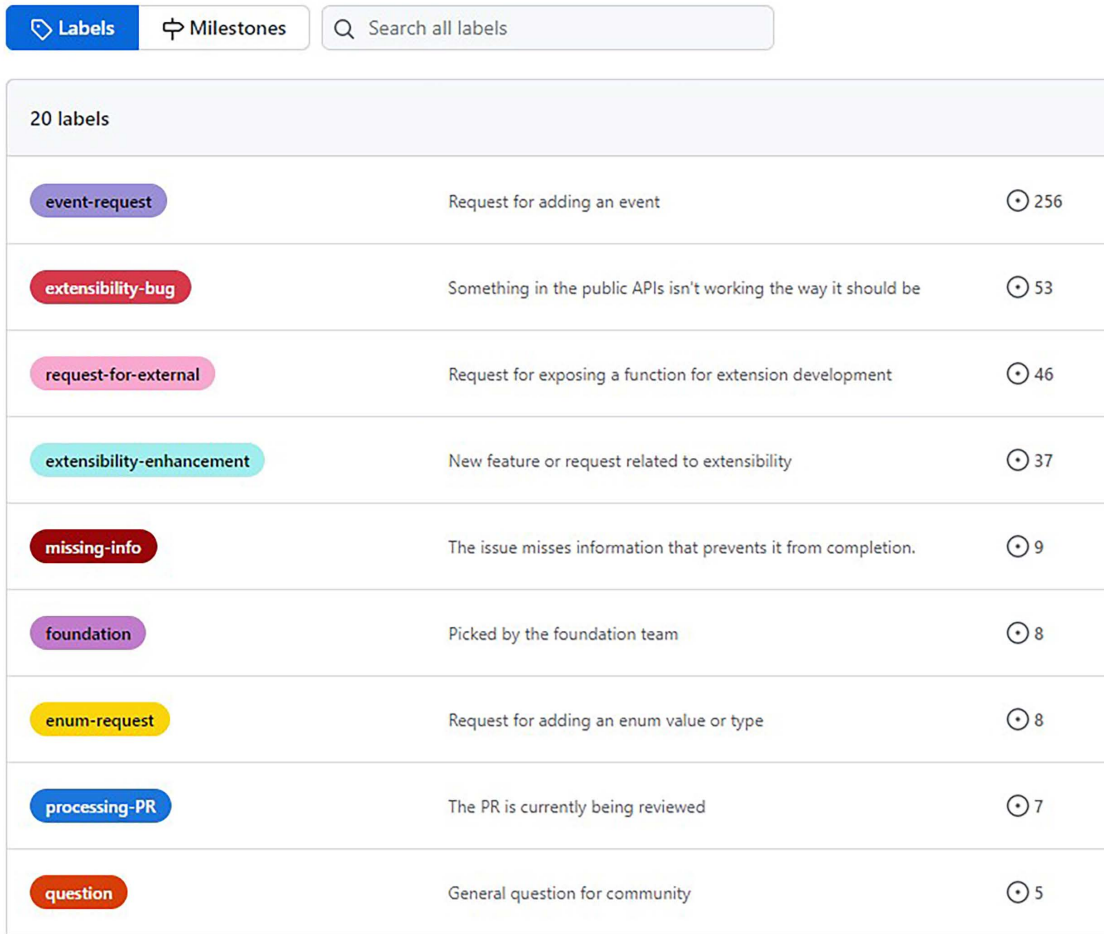
Dynamics 365 Business Central exposes a lot of events in its standard code and new events are added monthly, under partner requests.

You should request new events by going to the following link and filing a new item: <https://github.com/Microsoft/ALAppExtensions/issues>.

The above GitHub repository is officially intended for all Dynamics 365 Business Central partners who need to do the following:

- **Add new integration events:** Get the event you need to hook into a process. Marked [event-request].
- **Change function visibility:** For example, make a public function external or make a similar change so you can call it from your extension and reuse the business logic. Marked [request-for-external].
- **Replace an option with an enum:** Replace a specific option with an enum that supports your extension. Marked [enum-request].
- **Extensibility enhancements:** Request changes in the application code that will improve extensibility. Marked [extensibility-enhancement].

Also looking at the labels contained in this GitHub repository, you might see that there are quite a few more than the ones listed above. These are used for internal purposes by Microsoft or to mark every item for release (for example, `next minor` or `next major`):



20 labels		
event-request	Request for adding an event	256
extensibility-bug	Something in the public APIs isn't working the way it should be	53
request-for-external	Request for exposing a function for extension development	46
extensibility-enhancement	New feature or request related to extensibility	37
missing-info	The issue misses information that prevents it from completion.	9
foundation	Picked by the foundation team	8
enum-request	Request for adding an enum value or type	8
processing-PR	The PR is currently being reviewed	7
question	General question for community	5

Figure 3.1: Microsoft GitHub *AlAppExtensions* labels

The valuable takeaway for you in relation to event requests is that, if you are upgrading an existing customer from Dynamics NAV or hybrid versions of Dynamics 365 Business Central (October 2018 or Spring 2019 releases) with customizations, *you must carefully add a timeline in your upgrade plan* in relation to event requests and their subsequent standard implementation by Microsoft.

We can consider two practical examples for filing items in the above GitHub repository. Either of these scenarios could easily occur:

- Part of the code that you have customized using C/AL does not fit into any of the existing event subscribers in the current standard code base.
- Some existing standard functions used in C/AL are marked with the `[OnPrem]` attribute decorator.

In the first case, this is the main cause for having to request new event subscribers. In the latter, you would have to remove the OnPrem decorator from standard procedures. You can find labels for these issues in the GitHub repository.

In this section, we learned how events are the fundamental building blocks of every AL extension. In the next section, we will have an overview of the available AL objects and learn how to create them with the AL Language extension.

## Understanding the basics of AL

An extension of Dynamics 365 Business Central is written using the AL programming language. With AL, you can create new objects, extend standard objects, create custom business logic for your application, and much more. You create an extension for Dynamics 365 Business Central by using Visual Studio Code as your development environment and by using the AL extension for Visual Studio Code.

All Dynamics 365 Business Central functionalities are coded as either new objects or extensions of standard objects, and these objects are defined within files that have the `.al` file extension. Every `.al` file can define multiple objects, but this is not recommended.

Extensions are then compiled as `.app` package files, and these files are the final extensions that you will deploy in your final environment. When we use the term *deploy*, we are referring to four distinct phases:

1. Publication
2. Synchronization
3. Data upgrades
4. Installation

Once an extension is deployed, the end user will be able to benefit from it and extend the current environment with the feature that it adds.

At the time of writing, the following objects are available with the AL extension for Visual Studio Code:

- Tables and table extensions
- Pages and page extensions
- Reports and report extensions
- Enums and enum extensions
- Codeunits
- XMLports
- Queries
- Dotnet and control add-ins (JavaScript)
- Profiles and page customizations
- Interfaces
- Entitlements, permission sets, and permission set extensions

We will look at the main objects in detail in the following sections. Some of these objects (such as reports, page customizations, and control add-ins) will be covered in later chapters.

## Creating a new workspace

After installing the AL Language extension in Visual Studio Code, you can start a new AL project by going to **View | Command Palette** and selecting **AL:Go!**.

Visual Studio Code asks you for a folder in which it can create the project and then asks you to select the target platform (a version of Business Central). Select **12.0 Business Central 2023 release wave 2**:

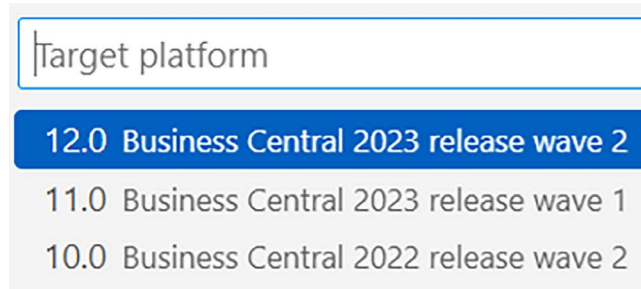


Figure 3.2: Selecting a target platform

Visual Studio Code will configure the project for you. It creates the `launch.json` file so that you can connect to your development environment and the `app.json` file with the extension's manifest file.

Both JSON files contain important configuration parameters used to define the extension and download/upload objects or symbols. A comprehensive list and definition of all parameters can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-json-files>.

Now, we can start defining the objects that comprise your solution.

## Defining objects using snippets

To declare and code AL objects, AL makes use of a native concept in Visual Studio Code called snippets, instead of you having to type every single character or line manually.

AL standard code snippets in Visual Studio Code are available after installing the AL extension. These are triggered as you type within the code editor, and you can recognize them by the square prefix symbol.

Typically, they start with the letter `t` and are followed by a meaningful name that describes what the snippet is about: for example, `ttable` or `tpage`. A tooltip shows a preview of the code snippet. The following screenshot shows the standard snippet for an `if-then-else` conditional sentence:

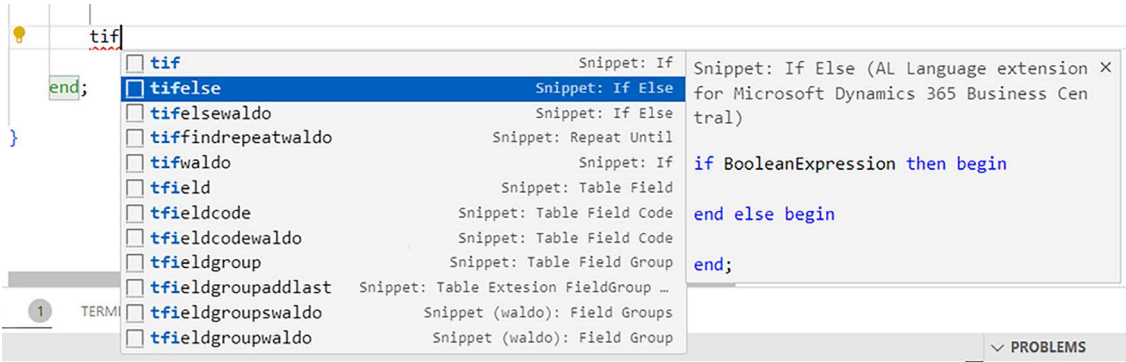


Figure 3.3: Searching for a code snippet

Notice that if the snippet contains variable names or code identifiers, they could be highlighted, suggesting that you should give them a different name and that they act as a sort of placeholder. When you rename a highlighted identifier, all occurrences will be also renamed, making snippet usage very flexible. This will not only reduce coding time, preventing the writing, or copying and pasting of repeated sentences, but it will also use the appropriate, complex syntax structure, which a developer might not keep in mind.

It is possible to download code snippets that have been produced by other developers in the form of extensions directly from Visual Studio Marketplace. Typically, many of the extensions that extend support for AL also include a series of code snippets of their own.

A typical example is **waldo's CRS AL Language extension**.

Together with several particularly useful developer tools, this extension also implements a set of AL code snippets that integrate with and enrich the existing standard ones. Currently, it implements 99 extra AL code snippets, and the list grows with every extension update.

The following screenshot shows all the available extra snippets that appear if you type `twaldo` in an `.al` file:

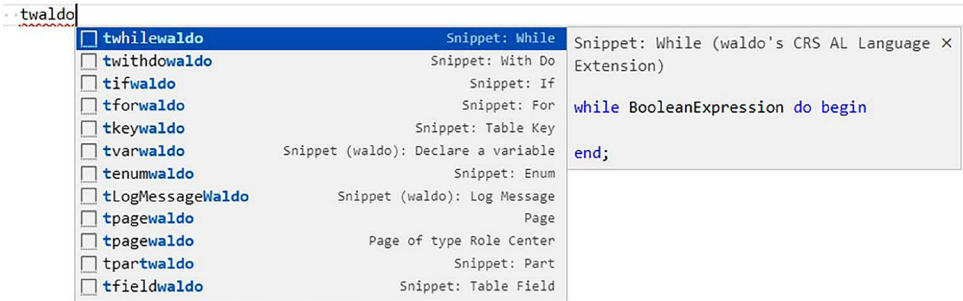


Figure 3.4: Extra twaldo snippets

Another way to search for code snippets while coding is to run the Command Palette (*Ctrl + Shift + P* or *F1*), type `snippets`, and choose to insert snippets, or simply press *Ctrl + Alt + J* to bring up a drop-down list of the AL snippets that are available.

If you still do not find the code snippets that are useful to you in the long list of extensions from the marketplace, with Visual Studio Code, it is also possible to manually add new code snippets from scratch. To accomplish this, you must click on the menu bar and go to **File (Alt + F) | Preferences (P) | Configure User Snippets** or run the Command Palette and search for `Configure User Snippets`.

You can then select whether to create a global snippet for all languages, a local snippet for the current workspace, or one that is specific to a target language. In this example, we will create a new snippet to be used with AL files by choosing **al.json (AL)** from the language list.

The following screenshot shows the available options when creating a specific code snippet:

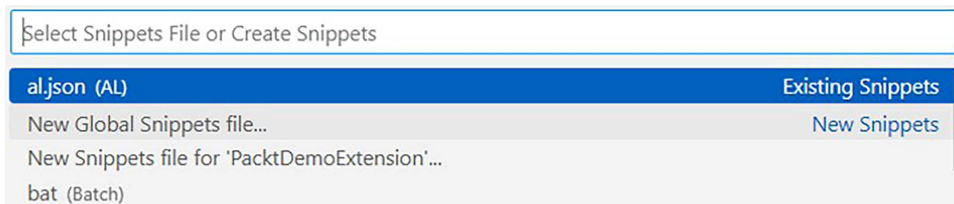


Figure 3.5: Snippet options

A specific configuration file is open in editing mode for custom AL snippets. Typically, the file is called `al.json`, and it is stored in the following location:

```
C:\Users\<username>\AppData\Roaming\Code\User\snippets
```

Each snippet is defined by a unique name and is composed of three elements:

- **Prefix:** Used to search and trigger the snippet in the editor
- **Body:** The section that is pasted inside the editor
- **Description:** A verbose description of what the snippet is for

Inside the body, you could use a specific syntax to enable placeholders:

- `$1`, `$2`, `$3`, ..., `$n` are used to move the cursor position within the snippet by pressing the *Tab* key.
- `$0` is used as the final cursor position.
- `${1:labelX}`, `${2:labelY}`, and `${3:labelZ}` are used as placeholders. Placeholders with the same IDs are connected to each other, enabling the multiple-cursors feature.

Now, we will go through a simple example. Imagine that you would like to add a standard code header block on top of the object, and you need a smart way to implement this repeatedly and quickly on every object.

The easiest solution is to create an ad hoc custom snippet to be invoked on every new object file creation, as follows:

1. Add the following code to the `al.json` file and save it:

```
{
  "Create standard comment block": {
    "prefix": "tcomment (Custom)",
    "body": [
      "//",
      "// ${1:YY.MM.DD} Initialization",
      "// ${1:YY.MM.DD} ${2:Modification Description}",
      "//"
    ],
    "description": "Standard header comment block"
  }
}
```



In Visual Studio Code, you can enable the amazing **autosave** feature by simply going to the **Menu** bar and selecting **File | Auto Save**. A checkmark will appear beside the **Auto Save** menu item. Another way to accomplish this task is to run the Command Palette (*Ctrl + Shift + P*) and type **File: Toggle Auto Save** (or type part of it and select the entry from the drop-down action list).

2. Create a new file inside any of your apps and save it with the `.al` extension (for example, `MyCodeunit.al`). The cursor should be automatically positioned in the first line and column of the file.
3. Start typing `tcomment` and IntelliSense will detect the existence of your custom snippet. Select it.
4. The cursor will be placed in the first placeholder element. Just type the current date in the `YY/MM/DD` format and press *Tab*. You might notice that since two placeholders share the same ID, they are edited together, enabling the multiple-cursors feature.
5. Now, it is time to write something useful that is related to the object description and what it is for.

These code snippets make it easy to understand Visual Studio Code. Try them out and master them! In the next pages, we will dissect the anatomy of AL objects, starting from their own standard snippet examples. The forthcoming list of object definitions will follow the typical order of creation when creating an extension.

## Table object definition

A table definition can be created by using the `ttable` snippet:

```
table id MyTable
{
    DataClassification = ToBeClassified;

    fields
    {
        field(1;MyField; Integer)
        {
            DataClassification = ToBeClassified;
        }
    }

    keys
    {
        key(PK; MyField)
        {
            Clustered = true;
        }
    }

    var
        myInt: Integer;

    trigger OnInsert()
    begin

    end;

    trigger OnModify()
    begin

    end;

    trigger OnDelete()
    begin

    end;
```

```

trigger OnRename()
begin

end;

}

```

To define a table, you need to specify an *ID* (which must be unique in your application extension, per object type) and a *name* (which must also be unique per application and object type). Then, you can set the table's properties (use the useful *Ctrl + Spacebar* operation to discover all the listed properties through the built-in IntelliSense tool):

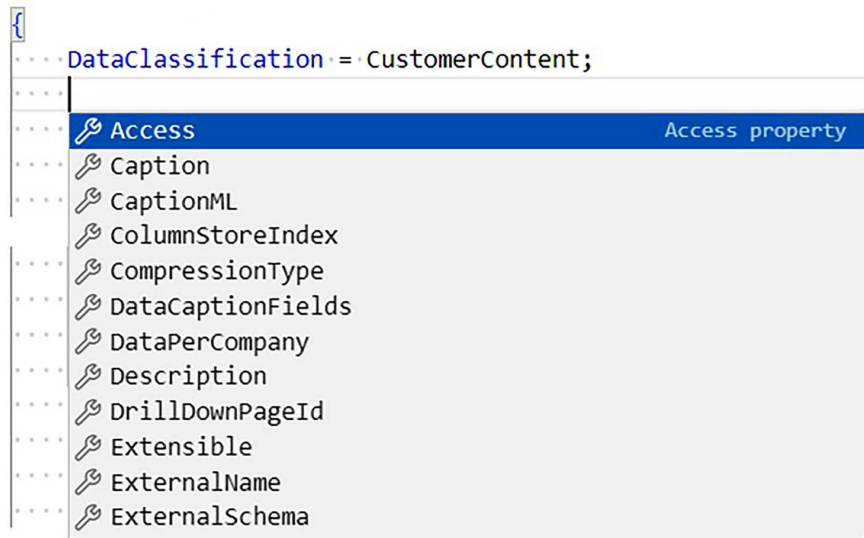


Figure 3.6: Setting table properties

A table object has the following main properties:

- **Caption:** The string that identifies the table in the user interface.
- **DataCaptionFields:** Sets the fields that appear to the left of the caption on pages that display the content of this table.
- **DataPerCompany:** Sets a value that indicates whether the table data applies to all the companies in the database or only the current company (when `default = true`, data is only available for the current company).
- **DrillDownPageID:** Sets the ID of the page to use as a drill-down.
- **LookupPageID:** Sets the ID of the page to use as a lookup.
- **LinkedObject:** Available for on-premises only; it specifies a link to a SQL Server object.
- **Permissions:** Sets whether an object has additional permissions that are required to perform some operations on one or more tables.

- **TableType:** Specifies the table type (Normal, CRM, ExternalSQL, Exchange, or MicrosoftGraph).
- **ExternalName:** This property appears when you specify CRM or ExternalSQL in the TableType property and specifies the name of the original table in the external database.
- **ExternalSchema:** This property appears when you specify CRM or ExternalSQL in the TableType property and specifies the name of the database schema in the external database.
- **ReplicateData:** Specifies whether the table must be replicated to the cloud (the default value is true).
- **Extensible:** Sets whether the object can be extended or not.

A table object contains a set of fields. A table's field can be created by using the `tfield` snippet:

```
field(id; MyField; Blob)
{
    DataClassification = ToBeClassified;
    FieldPropertyName = FieldPropertyValue;
}
```

A field is defined by an *ID* (which must be unique within the declaring table and all its extensions), a *name* (which must also be unique within the declaring table and all its extensions), and a *type* (the data type of the field).

It's recommended to always set the `Caption` property (for tables and fields) and to set the `DataClassification` property (used for defining the data sensitivity for GDPR regulations) to a value other than `ToBeClassified`. A field can have its own specific properties, which you can set as needed (optional properties, as shown in the following screenshot):

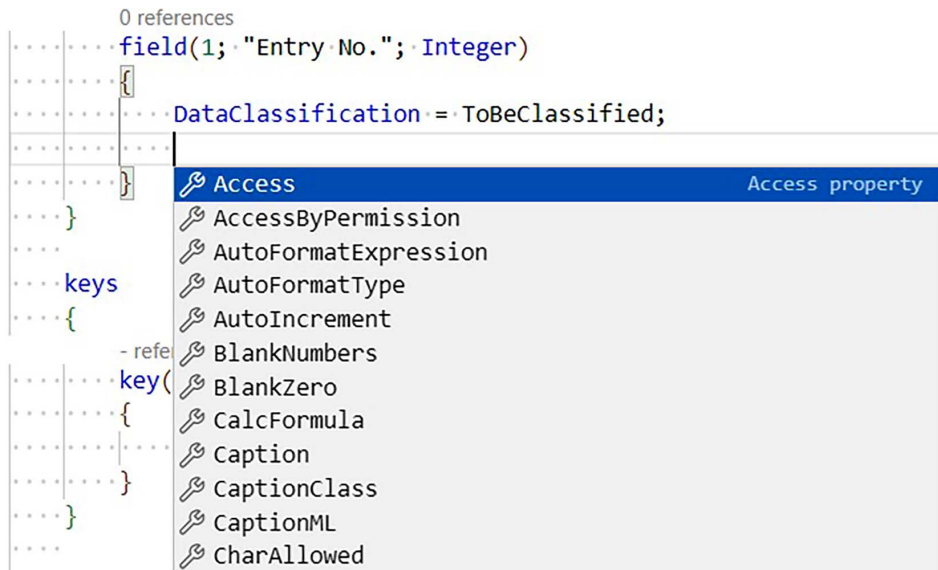


Figure 3.7: Field-specific properties

A table also contains a set of *keys*. You must have at least one key, and the key at the top of the list is automatically made the primary key. You can define keys using the `key` snippet:

```
key(MyKey; MyField)
{
}
}
```

A table's key is defined by a *name* and the *fields* that comprise the key (a comma-separated list of table fields). A key can have the `Clustered` property set to `true` if it is the primary key of the table. A clustered index is a special type of index that reorders the way the records in the table are physically stored, so a table can have only one clustered index.

A table can also have triggers (`OnInsert`, `OnModify`, `OnDelete`, and `OnRename`), and inside a table, you can define your own methods.

## Page object definition

A page object is the user interface for your users in Dynamics 365 Business Central. You can define a page object in AL using `page` snippets (we will learn more about this in the next chapter). AL provides four different built-in snippets, which allow you to create the following page types:

- A Card page
- An API page
- A List page
- A Role Center page

A Card page is defined as follows:

```
page Id MyPage
{
    PageType = Card;
    ApplicationArea = All;
    UsageCategory = Administration;
    SourceTable = TableName;

    layout
    {
        area(Content)
        {
            group(GroupName)
            {
                field(Name; NameSource)
                {
                    ApplicationArea = All;
                }
            }
        }
    }
}
```

```

    }
  }
}

actions
{
  area(Processing)
  {
    action(ActionName)
    {
      ApplicationArea = All;
      trigger OnAction()
      begin
        end;
    }
  }
}

var
  myInt: Integer;
}

```

A Card page, like all other pages and almost all objects, is identified by its *ID* and its *name* (both of which must be unique inside the application). A page also has its own properties and the main ones to define are as follows:

- **PageType:** Identifies the type of the page.
- **SourceTable:** Sets the underlying table for this page.
- **SourceTableView:** Sets the key, sort order, and filter you want to use to determine the view of the source table presented to the user.
- **ApplicationArea:** Sets the visibility of the page inside the Business Central application. At the page level, AppArea provides a default value for all controls on that page. The most common values are All, Basic, Suite, and Advanced.
- **UsageCategory:** Sets the Departments column for the searched page in the web client.
- **Extensible:** Sets whether the object can be extended or not.

A page has a layout area (which defines the page appearance in the UI) and an actions section (which defines the available menu items for adding code actions inside a page). Inside the layout, you have a content area, which contains a set of groups, and every group can contain one or more page fields. You can add a field inside a page group by using the `tpagefield` snippet:

```
field(MyField; FieldSource)
{
    ApplicationArea = All
    FieldPropertyName = FieldPropertyValue;
}
```

A field on a page is defined by a *name* (the field keyword inside the page) and a *field source* (the source expression of the page field, which corresponds to the physical fields defined in the underlying table). Remember that currently, a standard snippet's FieldSource does not implement the Rec. FieldSource syntax but this could soon be mandatory to avoid implicit issues. Also, remember that setting ApplicationArea at the field and/or action level is no longer necessary since the page-level value of ApplicationArea is inherited.

A List page is defined as follows:

```
page Id PageName
{
    PageType = List;
    ApplicationArea = All;
    SourceTable = TableName;

    layout
    {
        area(Content)
        {
            repeater(Group)
            {
                field(Name; NameSource)
                {
                    ApplicationArea = All;
                }
            }
        }

        area(Factboxes)
        {
        }
    }

    actions
    {
```

```

        area(Processing)
        {
            action(ActionName)
            {
                ApplicationArea = All;

                trigger OnAction();
                begin
                    // ...
                end;
            }
        }
    }
}

```

Applying this definition to an example, a List page would have the PageType property set to List, and the layout section would have Content and FactBox areas. The Content area would also have a repeater group, which would contain all the fields you want to display on that list. After that, you would have the actions section.

## Table extension object definition

As we mentioned previously, with Dynamics 365 Business Central, you cannot modify an existing table; instead, you need to create a table extension.

A table extension can be defined by using the `tableext` snippet:

```

tableextension Id MyExtension extends MyTargetTable
{
    fields
    {
        // Add changes to table fields here
    }

    var
        myInt: Integer;
    }
}

```

A `tableextension` object is defined by an *ID* and a *name* (which must be unique) and by the table name that must be extended. Then, inside the `fields` group, you can add new fields or change existing field properties.

The following code is an example of an extension to the standard Customer table that adds some new fields and changes an existing field property:

```
tableextension 50100 CustomerExtSD extends Customer
{
    fields
    {
        field(50100; PacktEnabledSD; Boolean)
        {
            DataClassification = CustomerContent;
            Caption = 'Packt Subscription Enabled';
        }
        field(50101; PacktCodeSD; Code[20])
        {
            DataClassification = CustomerContent;
            Caption = 'Packt Subscription Code';
        }

        modify("Net Change")
        {
            BlankZero = true;
        }
    }
}
```

In a tableextension object, you can also add new keys to the extended table by adding a group of keys, exactly in the same way you can do so for a table definition. For example, in our previous tableextension object, we have added two new fields, and we want also to create a secondary key for those fields in the Customer table. We can create a key group with the key name and key fields:

```
tableextension 50100 CustomerExtSD extends Customer
{
    fields
    {
        field(50100; PacktEnabledSD; Boolean)
        {
            DataClassification = CustomerContent;
            Caption = 'Packt Subscription Enabled';
        }
        field(50101; PacktCodeSD; Code[20])
        {
            DataClassification = CustomerContent;
            Caption = 'Packt Subscription Code';
        }
    }
}
```

```

    }
    modify("Net Change")
    {
        BlankZero = true;
    }
}

keys
{
    key(PacktKey; PacktCodeSD, PacktEnabledSD)
    {
    }
}
}

```

#### Current limitations:



- It is not possible to create a composite key based on fields from different extensions.
- It is not possible to modify, delete or alter an existing key in an extended table. It was announced with the 2023 release wave 2 (October 2023) that secondary keys would be allowed to be disabled, partially overcoming this limitation. This is still not part of the wave, but it should be implemented sooner rather than later.

Here, we have defined a secondary key called `PacktKey` in the `Customer` table, which consists of two custom fields (`PacktCodeSD` and `PacktEnabledSD`). Defining secondary keys is extremely useful for increasing the performance of some calculations, sorting records, and faster reporting.

Remember that not all the available table properties can be modified via a `tableextension` object. The list of table properties that can be modified is officially defined and maintained by Microsoft here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/properties/devenv-table-property-overview>.

## Page extension object definition

Exactly like tables, with Dynamics 365 Business Central, you cannot directly modify an existing page; instead, you need to create a page extension (using the `pageextension` object in AL).

A `pageextension` object can be defined by using the `tpageext` snippet:

```

pageextension Id MyExtension extends MyTargetPage
{
    layout
    {
    }
}

```

```

        // Add changes to page layout here
    }

    actions
    {
        // Add changes to page actions here
    }

    var
        myInt: Integer;
}

```

A pageextension object is defined by an *ID* and a *name* (which must be unique) and by the page that must be extended. A pageextension object contains a layout block (where you can add changes to the standard page layout, such as adding new fields or new sections or changing standard fields) and an actions block (where you can add your new actions).

The following is an example of a pageextension object in which we have added a new field to the Customer Card page (the field is added at the end of the General tab), and we have modified the Style property of an existing field (the Name field):

```

pageextension 50100 CustomerCardExtSD extends "Customer Card"
{
    layout
    {
        addlast(General)
        {
            field(PacktEnabledSD; PacktEnabledSD)
            {
                ApplicationArea = All;
            }
        }

        modify(Name)
        {
            Style = Strong;
        }
    }
}

```

As you can see, we have added a field to the page and modified the `Style` property of the `Name` field so that it will be displayed in bold at runtime. Remember that not all the available page properties can be modified via a `pageextension` object. The list of table properties that can be modified is officially defined and maintained by Microsoft here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/properties/devenv-page-property-overview>.

## Codeunit object definition

A codeunit is a container of AL code, and this code can be triggered by directly executing the codeunit (with the `OnRun` trigger) or by calling the procedures defined in the codeunit itself.

We can define a codeunit in AL by using the `tcodeunit` snippet:

```
codeunit Id MyCodeunit
{
    trigger OnRun()
    begin

    end;

    var
        myInt: Integer;
}
```

A codeunit is defined by an *ID* and a *name* (which must be unique inside your application). By default, the codeunit skeleton only contains the `OnRun` trigger definition, and inside this trigger, you can write the code that you want to execute when calling the `Codeunit.RUN` method.

It is worth mentioning that the `OnRun` trigger is not mandatory, so if you are not going to write code in `OnRun`, then you should delete the trigger from the codeunit and, obviously, also the automatically generated `myInt` global variable.

If you are going to write code in `OnRun`, then the standard codeunit snippet is going to be used quite often since you must always change the scope and you will always end up spending more time deleting useless elements; it is more efficient, then, to just type it out and let IntelliSense help you.

In a codeunit, you can define procedures (functions), which can be local to the codeunit or global (that is, publicly exposed to objects that instantiate the codeunit).

A procedure can be defined by using the `tpprocedure` snippet:

```
local procedure MyProcedure()
var
    myInt: Integer;
begin

end;
```

By default, this snippet creates a local procedure without parameters and without a return value. You can change the scope from local (the default value, meaning that it is visible only inside the object that declares the procedure) to global (so that it is also visible outside the object) by removing the `local` keyword.

As an example, this is a global procedure with parameters and a return value:

```
procedure CheckIfPacktCustomerIsEnabled(CustomerNo: Code[20]): Boolean
var
    //Local variables here
begin
    //Method code here
end;
```

A codeunit can have more than one procedure (local or global) defined.

## Event object definitions

As we mentioned previously, events are fundamental building blocks when it comes to developing extensions for Dynamics 365 Business Central.

When working with events, we have two main actors: the event *publisher* and the event *subscriber*.

An **event publisher** (an event that is raised by the application at runtime) can be defined in AL by using the `teventbus` (for a business event) or `teventint` (for an integration event) snippet. The main difference between business and integration events is that the first one defines a formal contract that will not change in future releases. For this reason, it is quite rare that in your daily job, you will write business events over integration events.

A **business event** has the following schema:

```
[BusinessEvent(IncludeSender)]
local procedure MyProcedure()
begin

end;
```

Here, `IncludeSender` is a Boolean value that specifies whether the global methods defined in the object that contains the event publisher method will be visible to the event subscriber methods that will subscribe to this event (this is `true` if the global methods must be visible and `false` – the default value – if not).

When the `IncludeSender` argument is set to `true`, the signature of the event subscriber methods that will subscribe to this published event will automatically include a `VAR` parameter (a reference value) for the published event object.

An **integration event** has the following schema:

```
[IntegrationEvent(IncludeSender,GlobalVarAccess)]
    local procedure MyProcedure()
    begin

    end;
```

Here, the IncludeSender Boolean parameter has the same meaning as we described previously.

GlobalVarAccess is a Boolean parameter that specifies whether the global variables defined in the object, which contains the event publisher method, are accessible to the event subscriber methods that subscribe to this published event (this is true if they must be exposed and false – which is the default value – if not).

When the GlobalVarAccess argument is set to true, all the event subscriber methods that subscribe to this event will be able to access the global variables in the object where the event publisher method is declared. You must manually add the variable parameters to the event subscriber methods, and you need to use a name and a type that match the variable declaration in the event publisher object.

After an event has been published by an event publisher (your previously defined method), you need to raise that event in your code where needed (event subscribers will not react to the event until it is raised in your application code).



Please note that allowing GlobalVarAccess is a bad practice and requires creating a lot of defensive code. This is the main reason Microsoft removed almost all instances of it from its base and system application.

As an example, the following is a codeunit with a public method that raises a business event and an integration event:

```
codeunit 50100 MyCodeunit
{
    procedure CheckIfPacktCustomerIsEnabled(CustomerNo: Code[20]): Boolean
    begin
        //Raising a business event
        MyBusinessEvent('XXX');

        //Other code here...

        //Raising an integration event
        MyIntegrationEvent('YYY');
    end;
```

```

[BusinessEvent(true)]
local procedure MyBusinessEvent(ID: Code[20])
begin
end;

[IntegrationEvent(true,true)]
local procedure MyIntegrationEvent(ID: Code[20])
begin
end;

//Global variables
var
    Customer: record Customer;
}

```

An **event subscriber** (a function that handles a raised event in the application) can be declared using the `teventsub` snippet:

```

[EventSubscriber(ObjectType::ObjectType, ObjectID, 'OnSomeEvent',
'ElementName', SkipOnMissingLicense, SkipOnMissingPermission)]
local procedure MyProcedure()
begin

end;

```

From the preceding code, we can see the following:

- `ObjectType` is an enum that identifies the object type that publishes the event to subscribe to (the object that contains the event publisher method) or that raises the trigger event to subscribe to.
- `ObjectID` is an integer value that specifies the ID of the object that publishes the event to subscribe to (when declaring it, do not use the ID; use the `ObjectType::Name` syntax).
- `OnSomeEvent` is a text parameter that specifies the name of the method that publishes the event in the object identified by the `ObjectID` parameter.
- `ElementName` is a text parameter that is used for database trigger events. It specifies the table field that the trigger event pertains to.
- `SkipOnMissingLicense` is a Boolean parameter that specifies what happens to the event subscriber method when the Dynamics 365 Business Central license of the user account that runs the current session does not include the permissions on the object that contains the subscriber method (true if the method call must be ignored and false if an error must be thrown and the code's execution must be stopped).

- `SkipOnMissingPermission` is a Boolean parameter that specifies what happens to the subscriber method when the user account that runs the current session does not have permission on the object that contains the event subscriber method (true if the method call must be ignored and false (the default value) if an error must be thrown and the code execution must be stopped).

As an example, this is a codeunit with two event subscribers for the business and integration events we defined in the previous example:

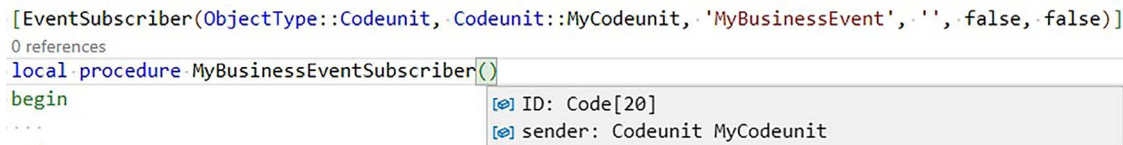
```
codeunit 50101 MySubscriberCodeunit
{
    [EventSubscriber(ObjectType::Codeunit, Codeunit::MyCodeunit,
    'MyBusinessEvent', '', false, false)]
    local procedure MyBusinessEventSubscriber(ID: Code[20])
    begin

    end;

    [EventSubscriber(ObjectType::Codeunit, Codeunit::MyCodeunit,
    'MyIntegrationEvent', '', false, false)]
    local procedure MyIntegrationEventSubscriber(ID: Code[20])
    begin

    end;
}
```

When defining the event subscriber, if you press *Ctrl + Spacebar* on the event parameters, you will see a list of the objects that the event can interact with (exposed by the publisher). In our example, the business event subscriber can see the event parameter and the sender object (because we have declared the event publisher with `IncludeSender` set to true), as follows:



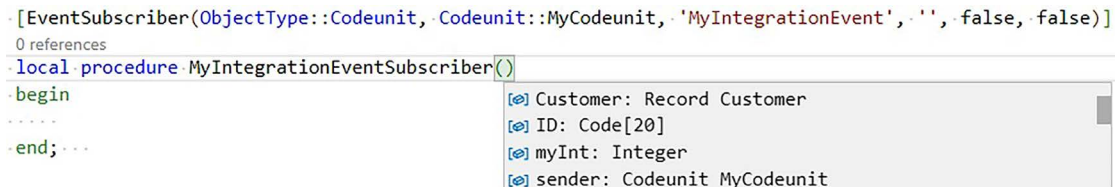
```
[EventSubscriber(ObjectType::Codeunit, Codeunit::MyCodeunit, 'MyBusinessEvent', '', false, false)]
0 references
local procedure MyBusinessEventSubscriber()
begin
    ...
end;
```

Expanded parameter list for `MyBusinessEventSubscriber`:

- ID: Code[20]
- sender: Codeunit MyCodeunit

Figure 3.8: Business event parameter and sender object

The integration event subscriber can see the event parameter, the sender object (because we have declared the event publisher with `IncludeSender` set to true), and the global variables of the sender object (because we have declared the event publisher with `GlobalVarAccess = true`):



```
[EventSubscriber(ObjectType::Codeunit, Codeunit::MyCodeunit, 'MyIntegrationEvent', '', false, false)]
0 references
local procedure MyIntegrationEventSubscriber()
begin
    ...
end;
```

Expanded parameter list for `MyIntegrationEventSubscriber`:

- Customer: Record Customer
- ID: Code[20]
- myInt: Integer
- sender: Codeunit MyCodeunit

Figure 3.9: Integration event parameter, sender object, and global variables

When using events, always remember the following:

- When the code that calls the event publisher method is run, all the event subscriber methods that subscribe to the event are run.
- If there are multiple subscribers, the subscriber methods are run one at a time in random order.
- If there are no subscribers to the published event, then the line of code that calls the event publisher method is ignored and not executed.

## XMLport object definition

XMLports are objects that are used for importing and exporting XML- or text-based data between an external source and Dynamics 365 Business Central.

An XMLport can be defined in AL by using the txmlport snippet:

```
xmlport Id MyXmlport
{
    schema
    {
        textelement(NodeName1)
        {
            tableelement(NodeName2; SourceTableName)
            {
                fieldattribute(NodeName3; NodeName2.SourceFieldName)
                {
                }
            }
        }
    }

    requestpage
    {
        layout
        {
            area(content)
            {
                group(GroupName)
                {
                    field(Name; SourceExpression)
                    {
                    }
                }
            }
        }
    }
}
```

```

    }
  }

  actions
  {
    area(processing)
    {
      action(ActionName)
      {
      }
    }
  }
}

var
  myInt: Integer;
}

```

As an example, this is a simple XMLport definition for importing some customer data (the No. and Name fields):

```

xmlport 50100 MyImportCustomer
{
  Direction = Import;
  schema
  {
    textelement(NodeName1)
    {
      tableelement(Customer; Customer)
      {
        fieldattribute(No; Customer."No.")
        {
        }
        fieldattribute(Name; Customer.Name)
        {
        }
      }
    }
  }
}
}

```

The XMLport object has the `Direction` property set to `Import` (only used for importing data to Dynamics 365 Business Central) and reads the `No` and `Name` fields from an XML object called `Customer`.

## Query object definition

A query object allows you to define an object that can be used to retrieve data from a single table or from multiple tables by applying filters and joins between tables. The returned result is a single dataset.

You can create a query in AL by using the `tquery` snippet:

```
query Id MyQuery
{
    QueryType = Normal;

    elements
    {
        dataitem(DataItemName; SourceTableName)
        {
            column(ColumnName; SourceFieldName)
            {
            }
            filter(FilterName; SourceFieldName)
            {
            }
        }
    }

    var
        myInt: Integer;

    trigger OnBeforeOpen()
    begin
    end;
}
```

As you can see, a query object has an `elements` section, and inside that section, you define a `dataitem` and the `column` elements that must be retrieved from it (the table fields to be included in the resulting dataset).

You can also create links between `dataitems` to retrieve data from more than one table.

As an example, the following is a query object that has been defined in AL so that it retrieves a list of customers, along with their sales and profit data:

```

query 50100 "Customer Overview"
{
    Caption = 'Customer Overview';
    elements
    {
        dataitem(Customer; Customer)
        {
            column(Name; Name)
            {
            }
            column(No; "No.")
            {
            }
            column(Sales_LCY; "Sales (LCY)")
            {
            }
            column(Profit_LCY; "Profit (LCY)")
            {
            }
            column(Country_Region_Code; "Country/Region Code")
            {
            }
            column(City; City)
            {
            }
            column(Salesperson_Code; "Salesperson Code")
            {
            }

            dataitem(Salesperson_Purchaser; "Salesperson/Purchaser")
            {
                DataItemLink = Code = Customer."Salesperson Code";
                column(SalesPersonName; Name)
                {
                }
                dataitem(Country_Region; "Country/Region")
                {
                    DataItemLink = Code = Customer."Country/Region Code";
                    column(CountryRegionName; Name)
                }
            }
        }
    }
}

```

```
{  
  {  
    {  
      {  
        {  
          {  
            {  
              {  
                {  
                  {  
                    {  
                      {  
                        {  
                          {  
                        }  
                      }  
                    }  
                  }  
                }  
              }  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

The query loops through the Customer table and then (for every customer) retrieves data from the other tables specified in the DataItemLink property.



Please note that queries return flattened datasets and not nested datasets. In this case, the application repeats the customer info for every salesperson record.

Query objects are extremely useful and powerful for retrieving records in your code. The first basic problem that you can solve with query objects is to avoid using nested loops when retrieving data from linked tables (joins). If you have Table1 linked through a foreign key to Table2, instead of looping through Table1 and, for every record of this table, going to Table2 to retrieve the related data, you can use a query object and apply the pattern described in the following diagram:

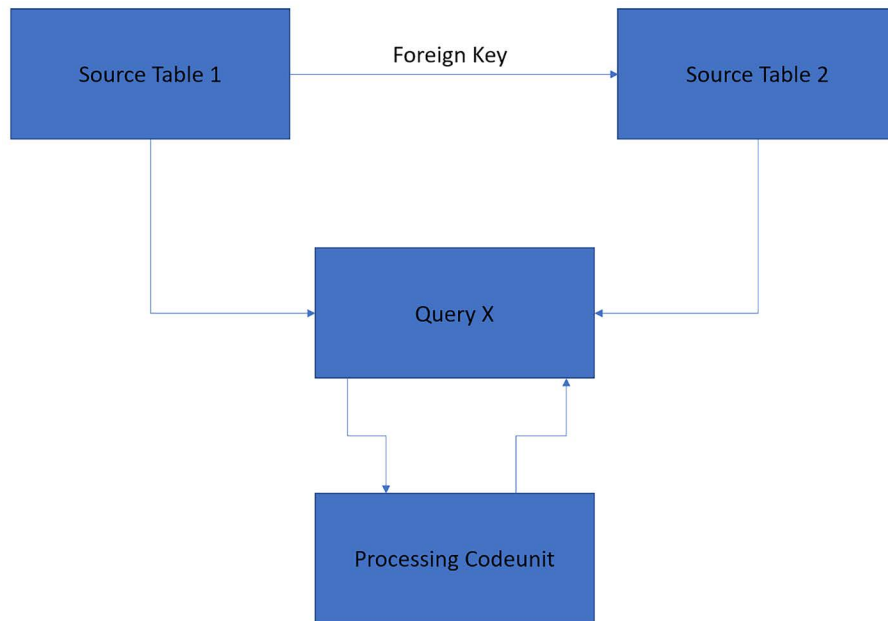


Figure 3.10: Pattern for using a query object and foreign key

Here, you can define a query that returns the fully filtered join of the two tables and then you can loop through the record set that is returned by the query object (this requires only one loop).

If (as an example) we want to use our previously defined Customer Overview query in our code, this is what we should do in AL:

```
procedure UseCustomerOverviewQuery()
var
    CustomerOverview: Query "Customer Overview";
begin
    if not CustomerOverview.Open() then
        exit;
    while CustomerOverview.Read() do
    begin
        //Here we have all joined records to loop
    end;
end;
```

Here, we execute the query object by calling the Open method, and then we loop through the returned dataset by using the Read method. Inside the loop, you have the complete record being returned by the query (the master table and the joined tables) and you can work on this data as needed.



If you are refactoring your solution or simply writing code, it is absolutely a good practice to think twice about whether you must create loops using the standard iterative AL statement, or if you can build an ad hoc query to improve performance.

## Enum object definition

Before we go into **enums**, let us talk briefly about their predecessors: **options**.

We will say off the bat that options should only be used when you are extending existing functionality that already uses options. Other than that, you should consider options to be outdated and you should only use enums for static value lists. However, it is important to know what option data types are, to understand enums.

A field with the option data type is used in Dynamics 365 Business Central to define a value from a predefined and static list of values. In practical terms, an option is an array of fixed values.

When you define an option field, you define the permissible values for that field in the following way, for example:

```
field(5; LicenseType; Option)
{
    OptionMembers = " ", "Full", "Limited";
    OptionCaption = ' ,Full,Limited';
    Caption = 'License Type';
    DataClassification = CustomerContent;
}
```

In the preceding code, we can see that the `OptionMembers` property contains the predefined value for the field. Here, the field named `LicenseType` contains three values (`blank`, `Full`, and `Limited`), and the first value (in this case, `blank`) is the default one.

But what if you want to extend these options, for example, by adding a new `LicenseType` field member called `Teams`? This is not possible. Option fields cannot be customized, of course not, nor extended.

Since options are not extensible, AL introduces a new object type named `enum`. An `enum` is an object type that consists of a set of named constants, and it can be extended from other extensions if its `Extensible` property is set to `true`, as shown here:

```
enum 50100 LicenseType
{
    Extensible = true;
    value(0; None) { }
    value(1; Full) { }
    value(2; Limited) { }
}
```

You can then define a field so that it has the `enum` type referenced in the following way:

```
field(50100; LicenseType; enum LicenseType)
{
    Caption = 'License Type';
    DataClassification = CustomerContent;
}
```

This allows you to define a field that has the same behavior as an option: when a user clicks on that field, Dynamics 365 Business Central presents a list of possible values to choose from.

To extend the `enum` field from another extension and add a new possible value called `Team`, you need to create an `enumextension` object, as follows:

```
enumextension 50110 LicenseTypeEnumExt extends LicenseType
{
    value(50110; Team)
    {
        Caption = 'Team License';
    }
}
```

After that, your `LicenseType` field will have one more `enum` value to choose from.

You can also use an enum object directly from AL code (as a variable):

```
var
    LicenseType: enum LicenseType;
begin
    case LicenseType of
        LicenseType::Full:
            //Write your code here...
```

You can also extend the TableRelation property of an enum value. For example, imagine you have the following table:

```
table 50120 LicenseDetail
{
    fields
    {
        field(1; Id; Integer) { }
        field(2; LicenseType; enum LicenseType) { }
        field(3; LicenseDetail; Code[20])
        {
            TableRelation =
                if (LicenseType = const (Full)) FullLicenseTable
                else if (LicenseType = const (Limited)) LimitedLicenseTable;
        }
    }
}
```

In this table, we have a field called LicenseType (which is an enum) and a field called LicenseDetail, which has a tableRelation property (to the FullLicenseTable and LimitedLicenseTable tables) based on the value of the enum field.

Another app could then extend both the enum field and the table relation so that it can handle the new extended enum. Below is an example:

```
enumextension 50110 LicenseTypeEnumExt extends LicenseType
{
    value(50110; Team)
    {
        Caption = 'Team License';
    }
}

tableextension 50110 LicenseDetailExt extends LicenseDetail
{
```

```

fields
{
    modify(LicenseDetail)
    {
        TableRelation = if (LicenseType = const (Team)) TeamLicenseTable;
    }
}
}

```

Here, the new app creates the `LicenseType` enum extension (as we described previously) and creates a new `tableextension` object, where it modifies the `TableRelation` property of the `LicenseDetail` field by adding a new relationship to a `TeamLicenseTable` if the enum has the value of `Team`.



The combined `TableRelation` is always evaluated from the top down, so the first unconditional relationship will prevail. This means that you cannot change an existing `TableRelation` from table A to table B if the original field has a relationship with table A.

By using `enum`, you can extend all your option's values. We recommend using this approach in your extensions if you want to enable extensibility.

But what if the Microsoft base application or any first-party extension still uses the option data type? As mentioned earlier, Microsoft listens, and for every release, it gathers partner requests to replace the fixed option fields in the standard code with extensible enums. The appropriate GitHub channel to place an enum change request is the same as where to request new event subscribers: <https://github.com/microsoft/ALAppExtensions/issues>.

Remember to add the `[Replace Option with Enum]` tag in the title to be more visible and easily searched for by you and the developer community.

## Profile object definition

A profile object allows you to define the user experience (main page) of a particular user profile. You can create a profile object with the AL Language extension by using the `tprofile` snippet.

A profile object is defined in the following example:

```

profile "SALES MANAGER"
{
    Caption = 'Sales Manager';
    ProfileDescription = 'Functionality for sales managers coordinating sales activities, such as tracking and assisting sales representatives to close sales and reporting on overall sales performance.';
    RoleCenter = 9005;
    Enabled = false;
}

```

Here, we have defined a profile called Sales Manager, which uses the RoleCenter page with ID = 9005 (a standard sales manager RoleCenter page object in Dynamics 365 Business Central).

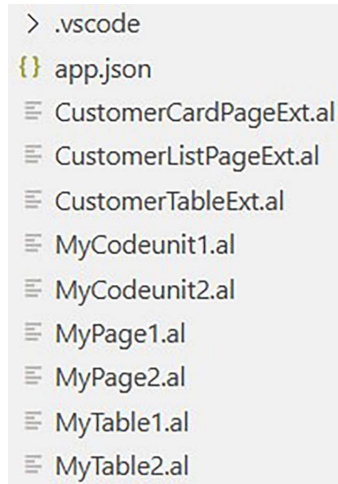
To deploy a profile object from your extension, I recommend creating a Profile folder in your AL project and, inside that folder, placing all the .al files that define your profiles.

In this section, you have had a complete overview of the more important available objects in the AL extension. In the next section, we will learn about some of the best practices when it comes to creating and handling an AL project.

## Understanding AL project structure best practices

As we mentioned previously, an AL project is file-based. You have all your .al files inside a project folder. The main problem that you encounter when you start working on a complex project is how to structure the project. How do we organize the objects and the .al files?

There is no written rule for this topic. What we could suggest is to avoid having all the objects (.al files) at the project root level, as shown in the following screenshot:



```
> .vscode
{} app.json
≡ CustomerCardPageExt.al
≡ CustomerListPageExt.al
≡ CustomerTableExt.al
≡ MyCodeunit1.al
≡ MyCodeunit2.al
≡ MyPage1.al
≡ MyPage2.al
≡ MyTable1.al
≡ MyTable2.al
```

*Figure 3.11: Root-level project structure*

Here, none of the objects are organized per business area, and they are only sorted by filename. If you have many objects, your object list will grow. In real-life scenarios, it will typically grow a lot, causing difficulties with handling and maintaining files.

The most sought-after way of structuring your project could be to organize your files by object type, as shown in the following screenshot:

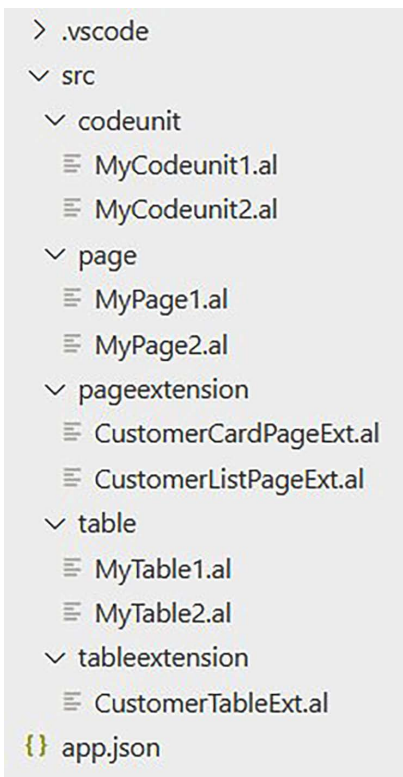


Figure 3.12: Object type project structure

Here, all the extension's code is inside the `src` folder. Then, all the objects are organized into each type, according to the objects that we have defined (there is a subfolder for every object type that we have in our solution). It is easier to find an object with this form of organization (just go to the object type folder), but this project structure has a drawback: it is not easy to recognize the objects that we need to implement a particular business functionality in our extension project.

Our suggestion, which Microsoft also typically does with its own extensions, is to try to organize your project tree first by functionality or business module and then by object type, as shown in the following screenshot:

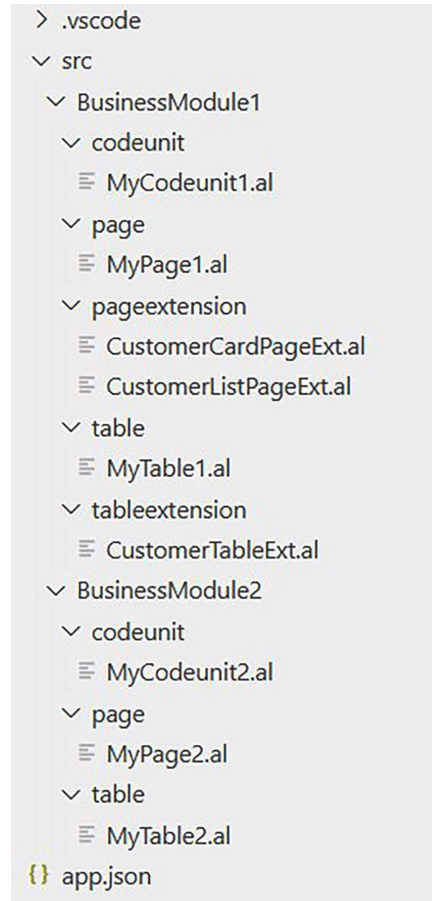


Figure 3.13: Module project structure

As you can see, in the `src` folder there are two subfolders: `BusinessModule1` and `BusinessModule2`. In these folders, objects are organized by their types. In the end, this is our recommended way of working, and this structure is extremely helpful in finding objects and provides an overview of the extension's purpose at the same time.



Note that there are other ways of organizing your files. You could, for example, use `Ctrl + P` in Visual Studio Code to find the file you need – in which case you could put everything in a single folder. Our recommendation is still to follow Microsoft's movement toward feature folders.

In the next section, we will learn how to name objects in AL and how to use object ranges.

## Naming guidelines and AL object ranges

When creating extensions for Dynamics 365 Business Central, you need to assign a numerical ID to your objects. This depends on whether you are creating a specific extension for a customer or intend to certify and deploy it in the official marketplace called **AppSource**: <https://appsource.microsoft.com/en-US/marketplace/apps?page=1&product=dynamics-365-business-central>.

Currently, with more than 4,000 extensions, *Dynamics 365 Business Central is by far the Microsoft service with the highest number of certified apps in its marketplace.*

The rules for assigning object IDs are as follows:

Range	Purpose
0 – 49,999	Microsoft-only. Reserved for standard features.
50,000 – 99,999	<b>Per-tenant extension range.</b> To customize the delivered solution to the individual needs of a customer.
100,000 – 999,999	Microsoft-only. Reserved for localized features.
1,000,000 – 69,999,999	<b>Registered Solution Program (RSP) range.</b> This was inherited by the old on-premises partner program. Do not request an object in these ranges unless you already have one already assigned.
70,000,000 – 74,999,999	<b>AppSource extensions.</b> This is the range that a partner should ask for if it needs to certify its solution and distribute it in one country or multiple ones worldwide.

Table 3.1: Assigning object IDs

Regarding file naming, each .al filename must start with the corresponding object type affix and object ID and must be written only with characters [A-Za-z0-9]. The file naming notation (which is mandatory for AppSource) should be as follows:

- **Full objects:** <ObjectNameExcludingAffix>.<FullTypeName>.al
- **Extension objects:** <ObjectNameExcludingAffix>.<FullTypeName>Ext.al

For each object type, you can use the following abbreviation (prefixes):

Object types	Abbreviation (affix)
Page, page extension	Page, PageExt
Table, table extension	Table, TableExt
Report, report extension	Report, ReportExt
Enum, enum extension	Enum, EnumExt
Codeunit	Codeunit
XMLport	XmlPort
Dotnet, control add-in	DotNet, ControlAddIn

Object types	Abbreviation (affix)
Profile, page customization	Profile, PageCust
Query	Query
Entitlement, permission set, permission set extension	Entitlement, Permissionset, PermissionsetExt
Interface	Interface

Table 3.2: Object type abbreviations

As an example, here are some AL objects and their corresponding filenames:

- Table 50,100, *Book*, should be called `Book.Table.al`
- Page 50,100, *Book Card*, should be called `BookCard.Page.al`
- Codeunit 50,110, *Book Management*, should be called `BookManagement.Codeunit.al`
- Page extension 50,101, *MyCustomerCardExt*, which extends *CustomerCard*, should be called `CustomerCard.PageExt.al`

You should also use an affix to target any of your object names. The affix will be reserved for you by Microsoft if you are developing for the AppSource marketplace; while developing per-tenant extensions, you can choose one your own, but it would be far better if you also registered an affix with Microsoft when you developed these to prevent naming conflicts with other AppSource extensions. This allows you to have objects that are named in a unique way between extensions, which avoids naming conflicts.

The rules for using the affix are as follows:

- The affix must be at least three characters long.
- The affix should reflect the general purpose of the extension for the application. You may want to include a partner's name when registering partner-specific affixes.
- Every object/field name must start or end with the affix.
- Using a table extension or a page extension, the affix must be defined at the control/field/action/group level.
- Use caption translation files (XLF) to handle multi-language labels in the UI.

For example, if you have reserved the PKT affix and you want to create a field called *CustomerCategory*, the valid field names that you could use are as follows:

- `PKT_ObjectName`
- `ObjectName_PKT`
- `PKT_ObjectName`
- `ObjectName_PKT`

It is strongly suggested to choose between the last two elements for better readability and to avoid conflicts if your affix could generate a meaningful field that might be used by any standard first party (this has happened in a real-life scenario).

If you opt for the third option, when sorting objects or fields, you have an immediate understanding of whether there are any custom ones and which ones they are. On the other hand, if you need to sort out all the objects, like when you pick them up from Visual Studio Code IntelliSense, this might be a drawback.

You can also do this with a suffix; just add the suffix to your search and all the relevant custom objects will show up. Visual Studio Code is great at this sort of fuzzy filtering. Type in, for example, `customer` and it will show the customer ledger.

The drawback of using prefixes is that now every custom object in your extension will be sorted under its prefix, not its name. Most likely, you would prefer objects to be sorted by their name, so it is worth keeping this in mind and using a suffix instead.

If you want to create a Customer Category table, the valid names for the table object could be as follows:

- `Table 70,000,000 PKT Customer Category`
- `Table 70,000,000 Customer Category PKT`
- `Table 70,000,000 PKT_Customer Category`
- `Table 70,000,000 Customer Category_PKT`

Using the reserved name as a prefix or a suffix is absolutely your choice.

These guidelines are mandatory for AppSource but are not mandatory for your per-tenant extensions. Our suggestion is to always follow these guidelines and take them as best practice in your daily developer life.

If you need to officially request and register an affix for your objects with Microsoft, you need to send an email to [d365val@microsoft.com](mailto:d365val@microsoft.com) specifying the name that you want to be reserved for your extension. Please use this email address only to request the registration of your objects and not for any other purpose.

## Working on AL coding guidelines

When creating your AL project (and your `.al` files), remember to always follow these main guidelines.

Inside an `.al` code file, the structure for all your objects must follow this sequence:

- Properties
- Object-specific constructs:
  - Table fields
  - Page layout
  - Actions
- Global variables:
  - Labels (old text constants)
  - Global variables
- Triggers
- Methods

Remember to always reference the AL objects by their object name and not by their ID. So, for example, this is how you reference a Record variable or a Page variable:

```
Vendor: Record Vendor;
Page.RunModal(Page::"Customer Card", ...);
```

In an event subscriber object, this is how you should reference the publisher object:

```
[EventSubscriber(ObjectType::Codeunit, Codeunit::MyCodeunit,
'MyIntegrationEvent', '', false, false)]
local procedure MyIntegrationEventSubscriber()
begin
end;
```

So, let us sum this up:

- **Format your AL code:** Take care of indentation and spacing (it keeps the code more readable). You can use *Alt + Shift + F* to auto-format your code.
- **Keep your .al files clean:** When using snippets, they automatically create an object skeleton with methods, properties, variables, triggers, or sections that you might not be using. Please remove all the code that is not being used. A typical example is trigger definitions on tables (which you can remove if you are not handling them) or global variables inside objects (if you do not remove them, your app will be full of `myInt : integer` variables).
- **Method declarations:** Be as local as possible. Only use global methods if you need to expose them to other objects.
- **Use events to trigger business logic.**
- **Design for extensibility:** Add event subscribers to your code in advance in order to let others subscribe to and extend the application.

We will cover more advanced development techniques in *Chapter 6, Advanced AL Development*. For now, just to have a flavor of more complex and advanced development, you can start using the **Generic Method** pattern:


- Declare each method on its class (table).
- Each method is a codeunit on its own (encapsulation).
- Invoke a method only from its class (table/codeunit).
- Each method's codeunit only has one global function.

- Local functions include the following categories (in this order):
  1. Main function (a method header in the form of a readable flowchart)
  2. Main business process (multiple functions)
  3. UI wrapper (two functions)
  4. Business extension (one or more functions to provide extensibility)
  5. Event wrapper (two functions)

This is an example of some AL code that has been organized according to this pattern:

<pre>codeunit 50113 CreatePalletMeth{   procedure CreatePallet (PurchLine: Record "Purchase Line"; HideDialog: Boolean);   begin     if not ConfirmCreatePallet(PurchLine, HideDialog) then exit;      OnBeforeCreatePallet(PurchLine, Handled);     DoCreatePallet(PurchLine, Handled);     OnAfterCreatePallet(PurchLine);      AckCreatePallet(PurchLine, HideDialog)   end;    local procedure DoCreatePallet(PurchLine: Record "Purchase Line"; var Handled: Boolean);    local procedure ErrorIfNotItemLine(PurchLine: Record "Purchase Line");   local procedure ErrorIfEmptyNumber(PurchLine: Record "Purchase Line");   local procedure DoInsertPallet(PurchLine: Record "Purchase Line");    local procedure ConfirmCreatePallet(PurchLine: Record "Purchase Line"; HideDialog: Boolean) : Boolean;   local procedure AckCreatePallet(PurchLine: Record "Purchase Line"; HideDialog: Boolean);    [BusinessEvent(false)]   local procedure OnBeforeCreatePallet(PurchLine: Record "Purchase Line"; var Handled: Boolean);SS   [BusinessEvent(false)]   local procedure OnAfterCreatePallet(PurchLine: Record "Purchase Line");    [IntegrationEvent(false,false)]   local procedure OnBeforeInsertPalletRecord(PurchLine: record "Purchase Line";var Pallet: record Pallet) }</pre>	Main
	Main Business Process
	UI Wrapper
	Event Wrapper
	Business Extension

Figure 3.14: Generic Method pattern



You can find more information regarding coding rules in the *Best Practices for AL* documentation article: <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/compliance/apptest-bestpracticesforalcode>.

Respecting coding rules and guidelines is extremely important for increasing code readability, and some of these rules are mandatory for AppSource.

Finally, considering development guidelines, the “voice” of the community is currently concentrated in one single repository. You can find out more about it from the following blog post, *Contributing to “ALGuidelines.Dev”*: <https://www.waldo.be/2021/12/03/contributing-to-alguidelines-dev/>.

An AL guidelines site was created a few years ago to have a general and global repository for several AL development pattern definitions, their guidelines, and best practices: <https://alguidelines.dev/>.



Figure 3.15: AL guidelines website

This is the result of a valuable open-source contribution, and you will find a substantial amount of information and best practices both from the Dynamics 365 Business Central community and Microsoft developers.

## Summary

In this chapter, we looked at the fundamentals of extension development with AL, along with an overview of the main objects for creating applications (tables, table extensions, pages, codeunits, and so on) and how to create them with Visual Studio Code.

It is important to always keep in mind the best practices when using these objects to build extensions. If you do not consider the organization of your project – the object IDs and naming conventions – then things will quickly get out of hand.

Supplementing this, always keep the guidelines for writing better code in the back of your mind. Format your code, keep your `.al` files clean, and focus on extensibility by adding event subscribers in advance to your code.

All in all, we learned how to create objects, how to create an AL project, how to handle its structure, and how to stick to naming conventions with objects.

In the next chapter, we will implement a real-world extension for Dynamics 365 Business Central by applying all these rules and best practices.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>





# 4

## Developing a Customized Solution for Dynamics 365 Business Central

In the previous chapter, we saw the fundamentals of extension development for Dynamics 365 Business Central and inspected all of the building blocks for creating extensions, such as basic object definitions, events, and how to extend standard objects (tables, pages, etc.).

This chapter focuses on creating a full extension for Dynamics 365 Business Central. We'll add new entities and functionalities to the platform, and we'll create a new extension following the best practices and code guidelines for compliance with Microsoft's AppSource marketplace.

This chapter will cover the following topics:

- Translating a business case into a real-world extension
- Writing code for extensibility
- Installing and upgrading extensions

### **Translating a business case into a real-world extension**

In this section, let's imagine having a Dynamics 365 Business Central customer with various business requirements. We want to create an extension to satisfy this customer's needs.

Our customer is a big commercial company that adopted Dynamics 365 Business Central as the company's ERP, and it has various business requirements that require customizations to be implemented.

The business requirements are the following:

Sales department:

- The company wants to classify customers based on custom categories that they can define as needed and that can change in the future. Each customer category must have its own details that can be used for some business processes inside the ERP.

- The sales office must be able to set up a default customer category and assign this default value to a customer automatically.
- The sales office needs the possibility to create gift campaigns for customer categories. A gift campaign is related to a limited period of time and a limited set of items.
- A gift campaign can be set to inactive for a certain period of time.
- When a gift campaign is active, the sales order manager must be able to automatically assign free gifts on a customer's sales order (they need a button on a sales order document that analyzes the order content, checks whether a campaign is active, and creates a line in the order for a free gift accordingly).
- When a sales operator inserts a sales order line, they should be alerted if the customer is ordering an item quantity near to the threshold of an active campaign promotion.
- When a sales order is posted, the generated item ledger entry record must store the customer category value that the customer has assigned at the time of posting for future reporting purposes.
- They want to have the possibility to calculate item charges based on defined criteria and automatically assign charges to a sales order by clicking a button on the sales order document.

#### Vendor quality management:

- The company has a quality process in place (CSQ, from the International Institute for the Certification of Business Quality) and they need to classify vendors according to their CSQ requirements:
  - Score related to item quality (from 1 to 10)
  - Score related to delivery on time (from 1 to 10)
  - Score related to item packaging (from 1 to 10)
  - Score related to pricing (from 1 to 10)
- The vendor quality card must also display some financial data:
  - Invoiced for current year N
  - Invoiced for the year *N-1*
  - Invoiced for the year *N-2*
  - Amount due for this vendor
  - Amount to pay (not already due) for this vendor
- The assigned scores determine a vendor rating (a numeric value) based on an algorithm.
- The purchase office cannot release a purchase order if the vendor does not meet standard company requirements (the vendor rating).
- The application's behavior should have the capacity to be extended in the future.

These customizations will be developed as a single extension by using the per-tenant object range (50.000 – 99.999). We will use AppSource best practices, and we'll use the PKT prefix (registered with Microsoft as our AppSource affix) to target all of our AL objects.



Affixes are used to uniquely identify an object between extensions in a given tenant. Affixes are mandatory for AppSource applications, not mandatory for per-tenant extensions (PTEs). We recommend always using affixes in your apps if you're developing PTEs.

## Developing the Dynamics 365 Business Central customization

We start our development tasks by opening Visual Studio Code and creating a new extension project (View | Command Palette | AL:GO!), selecting the latest Dynamics 365 Business Central release as the target platform.

We set the extension's manifest file (app.json) as follows:

```
{
  "id": "dd03d28e-4dfe-48d9-9520-c875595362b6",
  "name": "Packt Demo Extension",
  "publisher": "PACKT",
  "brief": "Customer Category, Gift Campaigns and Vendor Quality Management",
  "description": "Customer Category, Gift Campaigns and Vendor Quality
Management",
  "version": "1.0.0.0",
  "privacyStatement": "",
  "EULA": "",
  "help": "",
  "url": "http://www.packtpub.com",
  "logo": "./Logo/ExtLogo.png",
  "dependencies": [],
  "screenshots": [],
  "platform": "1.0.0.0",
  "application": "23.0.0.0",
  "features": [
    "NoImplicitWith", "TranslationFile"
  ],
  "idRanges": [
    {
      "from": 50100,
      "to": 50149
    }
  ],
  "contextSensitiveHelpUrl": "https://PacktDemoExtension.com/help/",
  "resourceExposurePolicy": {
```

```
    "allowDebugging": true,  
    "allowDownloadingSource": true,  
    "includeSourceInSymbolFile": true  
  },  
  "runtime": "12.0"  
}
```

Here, we set the extension details, such as the name, publisher, version, description, the path of the logo image, the admitted object range IDs (from 50100 to 50149), and the supported runtime version.

We also set the following option:

```
"features": [  
  "TranslationFile"  
]
```

The TranslationFile feature means that we want to have an XLIFF translation file that handles the multilanguage capabilities of this extension (Dynamics 365 Business Central is multi-language enabled, which means that you can display the user interface in different languages according to the user's setting).

We want to organize our project structure with subfolders for functionalities and then for object types. Our base project structure will be as follows:

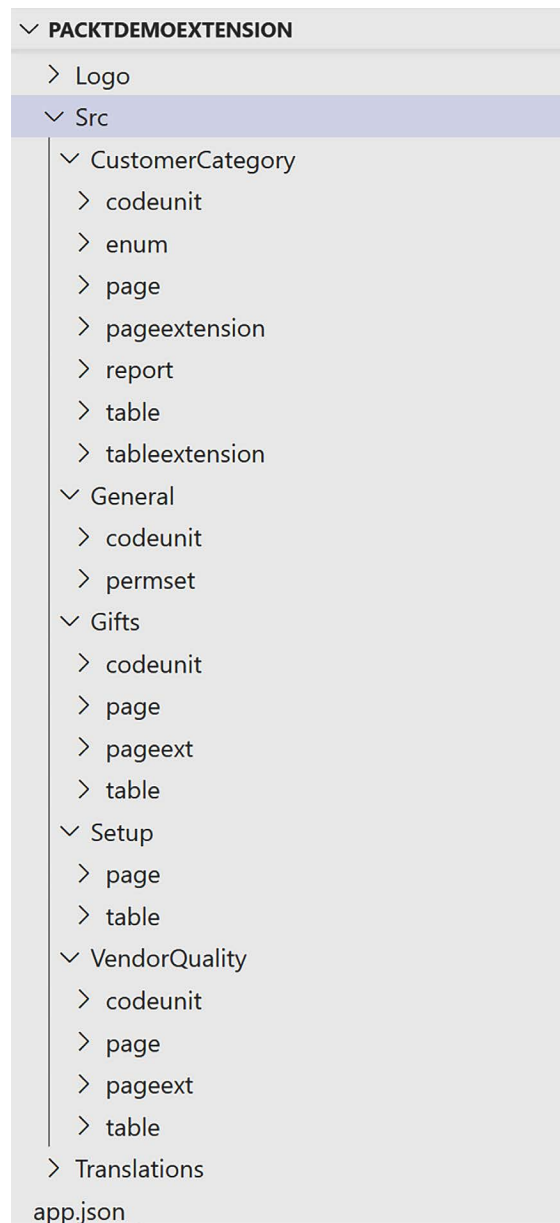


Figure 4.1: Project structure

Here, we have an Src folder, and inside that, we have three main folders for the following functionalities:

- **CustomerCategory:** This contains the implementation of the Customer Category requirements.
- **Gifts:** This contains the implementation of the gift campaign requirements.
- **VendorQuality:** This contains the implementation of the vendor quality requirements.

Inside each of these folders, we have subfolders organized into object types.



Please remember that this project structure is absolutely a personal choice. You can organize your objects and folders as you want; this is not a requirement for creating extensions in Dynamics 365 Business Central.

Let's start working on each of these three modules.

## Customer category implementations

One of the business requirements was to have the ability to group customers by category.

To handle the customer category management requirements, we need to do the following:

1. Define the Customer Category table.
2. Create the pages (user interface) that will handle the Customer Category entity (the List and Card pages).
3. Add a new Customer Category field to the standard Customer table.
4. Add the new field to the standard Customer Card page in the **General** tab and add some actions to the Customer pages to handle some tasks.
5. Create the business logic to handle the requirements.

In the next sections, we'll see the definitions and implementations of the various objects in detail.

### Table definition

By using the `ttable` and the `tfield` snippet, we define the Customer Category table as follows:

```
table 50100 "PKT Customer Category"
{
    Caption = 'Customer Category';
    DrillDownPageId = "PKT Customer Category List";
    LookupPageId = "PKT Customer Category List";
    DataClassification = CustomerContent;

    fields
    {
        field(1; Code; Code[20])
        {
            Caption = 'No.';
        }
        field(2; Description; Text[50])
        {
```

```

        Caption = 'Description';
    }
    field(3; Default; Boolean)
    {
        Caption = 'Default';
    }
    field(4; EnableNewsletter; Enum "PKT NewsletterType")
    {
        Caption = 'Enable Newsletter';
        DataClassification = CustomerContent;
    }
    field(5; FreeGiftsAvailable; Boolean)
    {
        Caption = 'Free Gifts Available';
    }
    field(6; Blocked; Boolean)
    {
        Caption = 'Blocked';
    }
    field(10; TotalCustomersForCategory; Integer)
    {
        Caption = 'No. of associated customers';
        FieldClass = FlowField;
        CalcFormula = count(Customer where("PKT Customer Category Code" =
field(Code)));
    }
}
keys
{
    key(PK; Code)
    {
        Clustered = true;
    }
    key(K2; Description)
    {
        Unique = true;
    }
}

procedure GetSalesAmount(): Decimal

```

```

var
    CustomerCategoryMgt: Codeunit "PKT Customer Category Mgt";
begin
    exit(CustomerCategoryMgt.GetSalesAmount(Rec.Code));
end;
}

```

The name of the object has the registered PKT prefix (to be unique across the application).

In this table definition, we have defined the following fields:

- **Code:** This is the code of the category (the key field).
- **Description:** This is the description of the category.
- **Default:** This is a Boolean field used to set the default category.
- **FreeGiftsAvailable:** This is a Boolean field used to set whether the category can be used with gift campaigns.
- **Blocked:** This is a Boolean field used to set the category as blocked (cannot be used).
- **EnableNewsletter:** This is an option field used to select the newsletter type to send for this category (commercial purposes). This field is of the enum type. As described in the previous chapter, the enum type allows us to have an extendable option field.
- **TotalCustomersForCategory:** This is a calculated field (flowfield) used to automatically calculate the number of customers associated with the selected category.

This table's definition has a key section, where we have defined a primary key (the No field) and a secondary key with the Description field. This secondary key is defined with the Unique property set to true, and this ensures that you cannot have records in this table with the same value as this field:

```

key(K2; Description)
{
    Unique = true;
}

```

The NewsletterType enum is defined as follows:

```

enum 50100 "PKT NewsletterType"
{
    Extensible = true;
    value(0; None)
    {
        Caption = 'None';
    }
    value(1; Full)
    {
        Caption = 'Full';
    }
    value(2; Limited)

```

```

    {
        Caption = 'Limited';
    }
}

```

As a generic programming rule, a table acts like a class and, in the table definition, we want to expose the methods related to that class. This is why we have defined a method (procedure) here called `GetSalesAmount` (which is used to return the total sales amount for the selected category).

The method's implementation will be on an external codeunit (which will contain our business logic).

We've also defined a setup table for this extension (the `Packt Extension Setup` table, which we'll also use in the next sections) to handle all of the variable parameters needed for the company's business configuration.

This setup table is defined as follows:

```

table 50103 "PKT Packt Setup"
{
    Caption = 'Packt Extension Setup';
    DataClassification = CustomerContent;

    fields
    {
        field(1; "Primary Key"; Code[10])
        {
        }
        field(2; "Minimum Accepted Vendor Rate"; Decimal)
        {
            Caption = 'Minimum Accepted Vendor Rate for Purchases';
        }
        field(3; "Gift Tolerance Qty"; Decimal)
        {
            Caption = 'Gift Tolerance Quantity for Sales';
        }
        field(4; "Default Charge (Item)"; Code[20])
        {
            Caption = 'Default Charge (Item)';
            TableRelation = "Item Charge";
        }
        field(10; "Shipmt Commission Calc. Method"; enum "PKT Shipm. Comm. Calc
Method")
        {

```

```

        Caption = 'Shipment Commission Calc. Method';
    }
}

keys
{
    key(PK; "Primary Key")
    {
        Clustered = true;
    }
}
}

```

It is best practice to have a dedicated setup table for an extension because it permits you to consolidate the settings in a single place. If you create an extension for the marketplace, we advise that you avoid adding setup settings to different standard Dynamics 365 Business Central setup tables if possible. If you want to do that, we recommend creating dedicated tabs instead.

## Page definition

To handle the Customer Category records (insert, modify, delete, and select), we need to have a list page and a card page.

By using the tpage snippet, we have defined a card page (PageType = Card) and a list page (PageType = List).

The list page (PKT Customer Category List) has an action for creating a default Customer Category record (it calls a method defined in an external codeunit because we don't want business logic on our pages).

The code for the list page definition is as follows:

```

page 50100 "PKT Customer Category List"
{
    Caption = 'Customer Category List';
    SourceTable = "PKT Customer Category";
    PageType = List;
    UsageCategory = Lists;
    ApplicationArea = All;
    CardPageId = "PKT Customer Category Card";
    AdditionalSearchTerms = 'ranking, categorization';
    AboutTitle = 'About Customer Categories';
    AboutText = 'Here you can define the categories for your customers. You
can then categorize your customers via the Customer Card.';
}

```

```
layout
{
    area(content)
    {
        repeater(Group)
        {
            field(Code; Rec.Code)
            {
            }
            field(Description; Rec.Description)
            {
            }
            field(Default; Rec.Default)
            {
            }
            field(TotalCustomersForCategory; Rec.TotalCustomersForCategory)
            {
                ToolTip = 'Total Customers for Category';
            }
        }
    }
}

actions
{
    area(processing)
    {
        action("Create Default Category")
        {
            Caption = 'Create default category';
            ToolTip = 'Create default category';
            Image = CreateForm;
            trigger OnAction();
            var
                CustManagement: Codeunit "PKT Customer Category Mgt";
            begin
                CustManagement.CreateDefaultCategory();
            end;
        }
    }
}
```

```

    }
  }
  area(Promoted)
  {
    group(PKTCustomerCategory)
    {
      Caption = 'Customer Category';
      actionref(CreateDefaultCategory; "Create Default Category")
      {
      }
    }
  }
}

```

As a best practice, to improve the search experience and help users to easily find the correct page by using the search feature of Dynamics 365 Business Central, we have also defined the `AdditionalSearchTerms` property. These terms will be used in addition to the `Caption` page property to find the page via the search engine.

Here, we've also defined a teaching tip for the page. Teaching tips can be defined at the page level and the control level and are useful to provide users a guided in-app tour experience explaining the functionalities of that page:

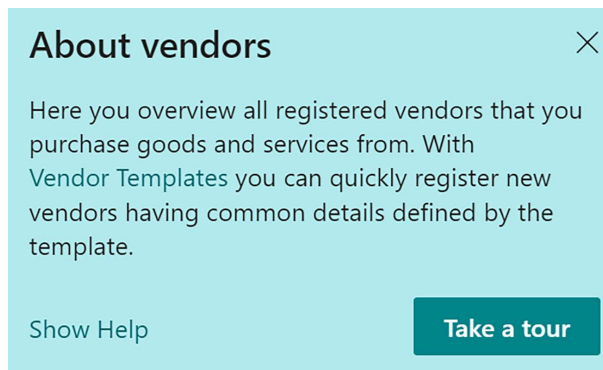


Figure 4.2: Teaching tips

Both types of teaching tips (page teaching tips and control teaching tips) are defined by using the `AboutTitle` and `AboutText` properties.

The page definition defines the promoted actions using the `actionref` syntax. We'll talk about that later in this chapter.

The **Customer Category List** page appears as follows:

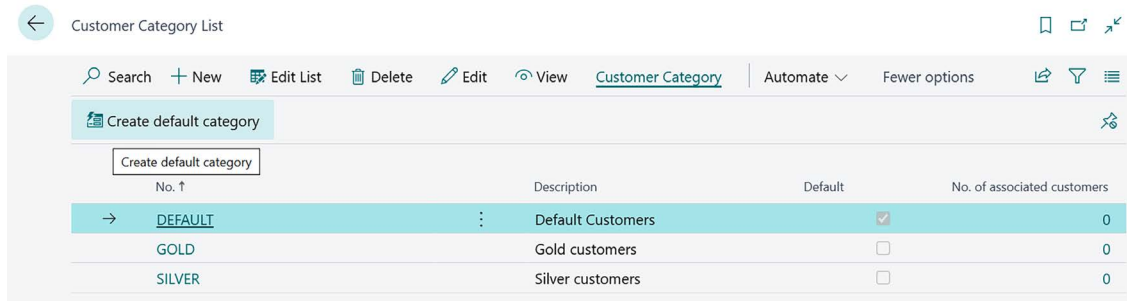


Figure 4.3: Page definition for the Customer Category List page

The card page (PKT Customer Category Card) has different groups for displaying data on separate FastTabs (General, Administration, and Statistics).

In the OnAfterGetRecord trigger, we calculate the total sales amount for the category, we assign that value to a global decimal field (called TotalSalesAmount), and we display this variable as a page field. The AL code defining the page is as follows:

```
page 50101 "PKT Customer Category Card"
{
    Caption = 'Customer Category Card';
    SourceTable = "PKT Customer Category";
    PageType = Card;
    ApplicationArea = All;
    UsageCategory = Documents;

    layout
    {
        area(Content)
        {
            group(General)
            {
                Caption = 'General';
                field(Code; Rec.Code)
                {
                }
                field(Description; Rec.Description)
                {
                }
                field(Default; Rec.Default)
                {
                }
            }
        }
    }
}
```

```

        field(EnableNewsletter; Rec.EnableNewsletter)
        {
        }
        field(FreeGiftsAvailable; Rec.FreeGiftsAvailable)
        {
        }
    }

    group(Administration)
    {
        Caption = 'Administration';
        field(Blocked; Rec.Blocked)
        {
        }
    }
    group(Statistics)
    {
        Caption = 'Statistics';
        field(TotalCustomersForCategory; Rec.TotalCustomersForCategory)
        {
            Editable = false;
        }
        field(TotalSalesAmount; TotalSalesAmount)
        {
            Caption = 'Total Sales Order Amount';
            Editable = false;
            Style = Strong;
        }
    }
}

trigger OnAfterGetRecord()
begin
    TotalSalesAmount := Rec.GetSalesAmount();
end;

var
    TotalSalesAmount: Decimal;
}

```

The Customer Category Card page looks like this:

←

Customer Category Card

✓ Saved

GOLD

General

No. .... GOLD

Enable Newsletter ..... Full

Description ..... Gold customers

Free Gifts Available .....

Default .....

Administration

Blocked .....

Statistics

No. of associated cust... 0

Total Sales Order Am... 0.00

Figure 4.4: Customer Category Card page

We’ve also created a page for the extension setup (called Packt Extension Setup), defined as follows:

```
page 50104 "PKT Packt Setup"
{
    Caption = 'Packt Extension Setup';
    PageType = Card;
    ApplicationArea = All;
    UsageCategory = Administration;
    SourceTable = "PKT Packt Setup";
    InsertAllowed = false;
    DeleteAllowed = false;

    layout
    {
        {
            area(Content)
            {
                group(General)
                {

```

```

        field("Minimum Accepted Vendor Rate"; Rec."Minimum Accepted
Vendor Rate")
        {
        }
        field("Gift Tolerance Qty"; Rec."Gift Tolerance Qty")
        {
        }
        field("Default Charge (Item)"; Rec."Default Charge (Item)")
        {
        }
    }
}

trigger OnOpenPage()
begin
    if not Rec.Get() then begin
        Rec.Init();
        Rec.Insert();
    end;
end;
}

```

This page looks like this:

← Edit + Delete ✓ Saved

## Packt Extension Setup

---

**General**

Minimum Accepted V...	6.00	Gift Tolerance Quantit...	0.00
-----------------------	------	---------------------------	------

Figure 4.5: Packt Extension Setup page

This will permit users to handle the settings for our extension.

## The tableextension definition

As per business requirements, we need to create a new field in the **Customer** table to handle the **Customer Category** assignment. In order to do that, we need to create a tableextension object, which can be done in AL by using the ttableext snippet.

The `tableextension` object for the **Customer** table is defined as follows:

```
tableextension 50100 "PKT CustomerExt" extends Customer
{
    fields
    {
        field(50100; "PKT Customer Category Code"; Code[20])
        {
            TableRelation = "PKT Customer Category";
            Caption = 'Customer Category Code';
            DataClassification = CustomerContent;

            trigger OnValidate()
            var
                CustomerCategory: Record "PKT Customer Category";
                BlockedErr: Label 'This category is blocked.';
            begin
                if CustomerCategory.Get("PKT Customer Category Code") then

                    if CustomerCategory.Blocked then
                        Error(BlockedErr);
                    end;
                end;
            end;
        }
    }

    keys
    {
        key(PKTCustomerCategory; "PKT Customer Category Code")
        {
        }
    }
}
```

Here, we've also handled the `OnValidate` trigger for this field to avoid the insertion of a blocked category.

We've also created a new secondary key on the **Customer** table based on this new field:

```
keys
{
    key(PKTCustomerCategory; "PKT Customer Category Code")
    {
    }
}
```

One of the requirements for this is to also add the **Customer Category Code** field to the **Item Ledger Entry** table (this must be written during posting for reporting purposes), so we have also defined the following tableextension object:

```
tableextension 50101 "PKT ItemLedgerEntryExt" extends "Item Ledger Entry"
{
    fields
    {
        //Field added during Sales Post
        field(50100; "PKT Customer Category Code"; Code[20])
        {
            TableRelation = "PKT Customer Category";
            Caption = 'Customer Category';
            DataClassification = CustomerContent;
        }
    }

    keys
    {
        key(PKTK1; "PKT Customer Category Code")
        {
        }
    }
}
```

This new custom field will be used for statistical purposes.

## The pageextension definition

This newly created Customer Category Code field must be visible on the Customer Card and Customer List pages.

To do that, we have defined two pageextension objects (by using the tpageext snippet). The following is the definition of the pageextension object for the Customer Card page:

```
pageextension 50102 "PKT CustomerCardExt" extends "Customer Card"
{
    layout
    {
        addlast(General)
        {
            field("PKT Customer Category Code"; Rec."PKT Customer Category
Code")
```

```

        {
            Tooltip = 'Customer Category';
            ApplicationArea = All;
        }
    }
}

actions
{
    addlast("F&unctions")
    {
        action("PKT Assign default category")
        {
            Image = ChangeCustomer;
            ApplicationArea = All;
            Caption = 'Assign Default Category';
            Tooltip = 'Assigns a Default Category to the current Customer';

            trigger OnAction();
            var
                CustomerCategoryMgt: Codeunit "PKT Customer Category Mgt";
            begin
                CustomerCategoryMgt.AssignDefaultCategory(Rec."No.");
            end;
        }
    }
}

```

This is the pageextension object definition for the Customer List page:

```

pageextension 50103 "PKT CustomerListExt" extends "Customer List"
{
    actions
    {
        addlast(Processing)
        {
            action("PKT Assign Default Category")
            {
                Image = ChangeCustomer;
                ApplicationArea = All;
            }
        }
    }
}

```

```

        Caption = 'Assign Default Category to all Customers';
        ToolTip = 'Assigns the Default Category to all Customers';

        trigger OnAction();
        var
            CustomerCategoryMgt: Codeunit "PKT Customer Category Mgt";
        begin
            CustomerCategoryMgt.AssignDefaultCategory();
        end;
    }
}
addlast(Promoted)
{
    group(PKTCustomerCategory)
    {
        Caption = 'Customer Category';
        actionref(PKTAssigningDefaultCategory; "PKT Assign Default
Category")
        {
        }
    }
}
}

views
{
    addlast
    {
        view(CustomersWithoutCategory)
        {
            Caption = 'Customers without Category assigned';
            Filters = where("PKT Customer Category Code" = filter(''));
        }
    }
}
}

```

Here, on the Customer List page, we've added an action to assign the category set by default for all customers. On Customer Card, the same action works on the currently selected record.

You can see here that these two functions call a method on an external codeunit named AssignDefaultCategory. This method has two implementations (it is overloaded), which we'll look at later in this chapter.

The standard Customer List page now looks like this:

Customers: All Search + New Manage Home New Document Customer Prices & Discounts Report Customer Category

Assign Default Category to all Customers

No. ↑	Name	Responsibility Center	Location Code	Phone No.	Contact	Balance (\$)	Balance
10000	Adatum Corporation				Robert Townes	0.00	
20000	Trey Research				Helen Ray	3,036.60	
30000	School of Fine Art				Meagan Bond	53,833.52	
40000	Alpine Ski House				Ian Deberry	4,316.92	
50000	Relecloud				Jesse Homer	8,836.80	

Figure 4.6: Customer List page

The Customer Card page looks like this:

Customer Card

10000 · Adatum Corporation

Home Request Approval New Document Prices & Discounts Customer Report Actions Related Reports Automate Fewer options

Assign Default Category Contact Apply Template Merge With... Send Email

General

No. 10000

Responsibility Center

Name Adatum Corporation

Document Sending Profile

IC Partner Code

Total Sales 223,598.40

Balance (\$) 0.00

Costs (\$) 40,255.70

Balance (\$) As Vendor 0.00

Profit (\$) 20,417.10

Balance Due (\$) 0.00

Profit % 33.7

Credit Limit (\$) 0.00

Last Date Modified 11/25/2022

Blocked

Disable Search by Name

Privacy Blocked

Customer Category Code

Salesperson Code JO

Address & Contact

Address

192 Market Square

Phone No.

Mobile Phone No.

Address 2

Email robert.townes@contoso.com

Country/Region Code US

Home Page

City Atlanta

Contact

Details

Attachments (0)

Customer Picture

Sell-to Customer Sales History

Customer No. 10000

Ongoing Sales Quotes 0	Ongoing Sales Market Orders 0	Ongoing Sales Orders 2
Ongoing Sales Invoices 2	Ongoing Sales Return Orders 0	Ongoing Sales Credit Memos 0
Posted Sales Shipments 33	Posted Sales Invoices 33	Posted Sales Return Receipts 0

Figure 4.7: Customer Card page

Here, we have the newly added Customer Category Code field and the new action for assigning the default category.

Codeunit definition

To handle the customer category business requirements, all of the required business logic is defined in a dedicated codeunit object called PKT Customer Category Mgt.

The codeunit is defined as follows:

```
codeunit 50100 "PKT Customer Category Mgt"
{
    procedure CreateDefaultCategory()
    var
        CustomerCategory: Record "PKT Customer Category";
        DefaultCodeTxt: Label 'DEFAULT';
        DefaultDescriptionTxt: Label 'Default Customer Category';
    begin
        CustomerCategory.Code := DefaultCodeTxt;
        CustomerCategory.Description := DefaultDescriptionTxt;
        CustomerCategory.Default := true;
        if CustomerCategory.Insert then;
    end;

    procedure AssignDefaultCategory(CustomerCode: Code[20])
    var
        Customer: Record Customer;
        CustomerCategory: Record "PKT Customer Category";
    begin
        //Set default category for a Customer
        Customer.Get(CustomerCode);
        CustomerCategory.SetRange(Default, true);
        if CustomerCategory.FindFirst() then begin
            Customer.Validate("PKT Customer Category Code", CustomerCategory.
Code);
            Customer.Modify();
        end;
    end;

    procedure AssignDefaultCategory()
    var
        Customer: Record Customer;
        CustomerCategory: Record "PKT Customer Category";
    begin
        //Set default category for ALL Customer
        CustomerCategory.SetRange(Default, true);
        if CustomerCategory.FindFirst() then begin
            Customer.SetFilter("PKT Customer Category Code", '%1', '');
            Customer.ModifyAll("PKT Customer Category Code", CustomerCategory.
Code, true);
```

```

        end;
    end;

    //Returns the number of Customers without an assigned Customer Category
    procedure GetTotalCustomersWithoutCategory(): Integer
    var
        Customer: Record Customer;
    begin
        Customer.SetRange("PKT Customer Category Code", '');
        exit(customer.Count());
    end;

    procedure GetSalesAmount(CustomerCategoryCode: Code[20]): Decimal
    var
        SalesLine: Record "Sales Line";
        Customer: Record Customer;
        TotalAmount: Decimal;
    begin
        Customer.SetCurrentKey("PKT Customer Category Code");
        Customer.SetRange("PKT Customer Category Code", CustomerCategoryCode);
        if Customer.FindSet() then
            repeat
                SalesLine.SetRange("Document Type", SalesLine."Document
Type":::Order);
                SalesLine.SetRange("Sell-to Customer No.", Customer."No.");
                SalesLine.SetLoadFields("Line Amount");
                if SalesLine.FindSet() then
                    repeat
                        TotalAmount += SalesLine."Line Amount";
                    until SalesLine.Next() = 0;
                until Customer.Next() = 0;

                exit(TotalAmount);
            end;
        }
    }

```

Here, we have the following functions:

- **CreateDefaultCategory:** This creates an entry in the Customer Category table with some predefined code and with the Default flag set to true.

- **AssignDefaultCategory:** This assigns the default category to customers. Here, we use overloading (supported in AL) and we have the same function with the following two different implementations (one without parameters and one with a Code[20] parameter):
  - `AssignDefaultCategory(CustomerCode: Code[20]):` Works on the current customer
  - `AssignDefaultCategory():` Works on all customers
- **GetTotalCustomersWithoutCategory:** This returns the number of customers without a category assigned.
- **GetSalesAmount:** This returns the total amount of the sales order for the customer category selected.

## Handling event subscribers

One of the business requirements for the customer category implementation is to block the posting of a sales order if a customer belongs to a category that is blocked.

We can do that by using event subscribers (see the `teventsub` snippet) and subscribe to the `OnBeforeCheckCustBlockage` event of the `Sales-Post` codeunit:

```
[EventSubscriber(ObjectType::Codeunit, Codeunit::"Sales-Post",
OnBeforeCheckCustBlockage, '', false, false)]
    local procedure OnBeforeCheckCustBlockageByCategory(SalesHeader: Record
"Sales Header"; CustCode: Code[20]; var ExecuteDocCheck: Boolean; var
IsHandled: Boolean; var TempSalesLine: Record "Sales Line" temporary)
    var
        Customer: Record Customer;
        CustomerCategory: Record "PKT Customer Category";
        CustBlockByCategoryLbl: Label 'Customer %1 is blocked by category';
    begin
        IsHandled := true;
        if Customer.Get(CustCode) then begin
            if Customer."PKT Customer Category Code" <> '' then begin
                CustomerCategory.Get(Customer."PKT Customer Category Code");
                if CustomerCategory.Blocked then
                    Error(CustBlockByCategoryLbl, Customer."No.");
            end;
        end;
    end;
```

As a best practice, we want to define event subscribers in a separate single-instance codeunit. A single-instance codeunit is an implementation of the *singleton* design pattern and only one instance of this object will be active per user session.

The codeunit and the event subscriber to handle the business requirement will be defined as follows:

```
codeunit 50103 "PKT General Event Subscribers"
{
    SingleInstance = true;

}
```

After this, we move on to implementing the gift campaign's business requirements.

## Gift campaign implementations

To handle the gift campaign requirements, we need to do the following:

1. Define the Gift Campaign table. This table must be able to store data as follows:

Customer Category	Item	Start Date	End Date	Minimum Quantity Ordered	Gift Quantity
GOLD	ITEM1	01/01/2019	30/03/2019	5	1
GOLD	ITEM2	01/01/2019	30/03/2019	10	2
SILVER	ITEM1	01/01/2019	30/03/2019	7	1

Table 4.1: Gift Campaign data fields

2. Create the page (user interface) for handling the gift campaign data (a list page).
3. Handling the business logic for assigning gifts to a sales order is based on Customer Category and the active campaign for this category. This will be done in an external codeunit.
4. Add a new function to the Sales Order page interface in order to permit the sales operator to automatically insert a gift line when the sales order is finished.
5. When the sales operator inserts the Quantity in a sales order line, we want to check the active campaigns and alert the user if the ordered quantity is close to the threshold of an active promotion.

## Table definition

By using the ttable snippet, we define the Gift Campaign table as follows:

```
table 50101 "PKT Gift Campaign"
{
    Caption = 'Gift Campaign';
    DataClassification = CustomerContent;
    DrillDownPageId = "PKT Gift Campaign List";
    LookupPageId = "PKT Gift Campaign List";

    fields
```

```
{
    field(1; CustomerCategoryCode; Code[20])
    {
        Caption = 'Customer Category Code';
        TableRelation = "PKT Customer Category";

        trigger OnValidate()
        var
            CustomerCategory: Record "PKT Customer Category";
            NoGiftsErr: Label 'This category is not enabled for Gift
Campaigns.';
            BlockedErr: Label 'This category is blocked.';
        begin
            CustomerCategory.Get(CustomerCategoryCode);
            if CustomerCategory.Blocked then
                Error(BlockedErr);
            if not CustomerCategory.FreeGiftsAvailable then
                Error(NoGiftsErr);
        end;
    }
    field(2; ItemNo; Code[20])
    {
        Caption = 'Item No.';
        TableRelation = Item;
    }
    field(3; StartingDate; Date)
    {
        Caption = 'Starting Date';
    }
    field(4; EndingDate; Date)
    {
        Caption = 'Ending Date';
    }
    field(5; MinimumOrderQuantity; Decimal)
    {
        Caption = 'Minimum Order Quantity';
    }
    field(6; GiftQuantity; Decimal)
    {
        Caption = 'Free Gift Quantity';
    }
}
```

```

    }
    field(7; Inactive; Boolean)
    {
        Caption = 'Inactive';
    }
}

keys
{
    key(PK; CustomerCategoryCode, ItemNo, StartingDate, EndingDate)
    {
        Clustered = true;
    }
}
}

```

The primary key for this table is a composite key, defined as follows:

```

key(PK; CustomerCategoryCode, ItemNo, StartingDate, EndingDate)
{
    Clustered = true;
}

```

Here, we've handled the OnValidate trigger of the CustomerCategoryCode field, which performs some validations:

- If the customer category selected is blocked, an error is thrown.
- If the customer category selected is not available for gift promotions (FreeGiftsAvailable = false), then an error is thrown.

Now our Gift Campaign table definition is ready and we can implement the user interface (page).

## Page definition

By using the tpage snippet, we define the Gift Campaign List page as follows:

```

page 50103 "PKT Gift Campaign List"
{
    Caption = 'Gift Campaigns';
    SourceTable = "PKT Gift Campaign";
    PageType = List;
    UsageCategory = Lists;
    ApplicationArea = All;
    AdditionalSearchTerms = 'promotions, marketing';
    AboutTitle = 'About Gift Campaigns';
}

```

```
AboutText = 'Here you can define the **Gift Campaigns** for your customers.  
With a gift campaign you can define promotional periods for your items and  
define gifts that a customer will receive when ordering some items.';
```

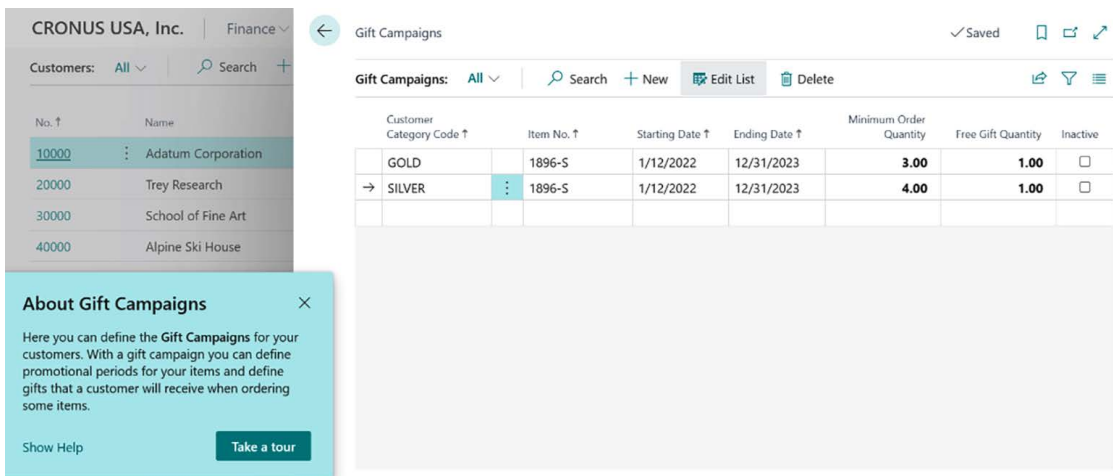
```
layout  
{  
    area(content)  
    {  
        repeater(Group)  
        {  
            field(CustomerCategoryCode; Rec.CustomerCategoryCode)  
            {  
            }  
            field(ItemNo; Rec.ItemNo)  
            {  
            }  
            field(StartingDate; Rec.StartingDate)  
            {  
            }  
            field(EndingDate; Rec.EndingDate)  
            {  
            }  
  
            field(MinimumOrderQuantity; Rec.MinimumOrderQuantity)  
            {  
                Style = Strong;  
            }  
            field(GiftQuantity; Rec.GiftQuantity)  
            {  
                Style = Strong;  
            }  
            field(Inactive; Rec.Inactive)  
            {  
            }  
        }  
    }  
}
```

```

views
{
    view(ActiveCampaigns)
    {
        Caption = 'Active Gift Campaigns';
        Filters = where(Inactive = const(false));
    }
    view(InactiveCampaigns)
    {
        Caption = 'Inactive Gift Campaigns';
        Filters = where(Inactive = const(true));
    }
}

```

When published, the page appears as follows, together with the relative teaching tip (opened the first time the user has access to this page):



The screenshot shows the 'Gift Campaigns' page in Microsoft Dynamics 365. The page header includes 'CRONUS USA, Inc.' and 'Finance'. The left sidebar shows a list of customers with columns 'No.' and 'Name'. The main area displays a table of gift campaigns with columns: Customer Category Code, Item No., Starting Date, Ending Date, Minimum Order Quantity, Free Gift Quantity, and Inactive. A teaching tip is displayed on the left side of the page.

**About Gift Campaigns**

Here you can define the Gift Campaigns for your customers. With a gift campaign you can define promotional periods for your items and define gifts that a customer will receive when ordering some items.

Show Help Take a tour

Customer Category Code	Item No.	Starting Date	Ending Date	Minimum Order Quantity	Free Gift Quantity	Inactive
GOLD	1896-S	1/12/2022	12/31/2023	3.00	1.00	<input type="checkbox"/>
→ SILVER	1896-S	1/12/2022	12/31/2023	4.00	1.00	<input type="checkbox"/>

Figure 4.8: Gift Campaigns page with teaching tip

To handle the gift assignment logic in a sales order, by creating a pageextension object, we have added a new action to the Sales Order page, and from this action, we call the AddGifts method defined in a codeunit in the next section.

The pageextension object is defined as follows:

```

pageextension 50100 "PKT SalesOrderExt" extends "Sales Order"
{
    actions
    {
        addlast(Processing)
    }
}

```

```
{
    action("PKT Add Free Gifts")
    {
        Caption = 'Add Free Gifts';
        ToolTip = 'Adds Free Gifts to the current Sales Order based on
active Campaigns';
        ApplicationArea = All;
        Image = Add;

        trigger OnAction()
        var
            GiftManagement: Codeunit "PKT Gift Management";
        begin
            GiftManagement.AddGifts(Rec);
        end;
    }
}
addlast(Promoted)
{
    group(PKTGifts)
    {
        Caption = 'Gifts';
        actionref(PKTAddFreeGifts; "PKT Add Free Gifts")
        {
        }
    }
}
}
```

The Sales Order page (with the new action) now looks like this:

Sales Order

S-ORD101005 · Adatum Corporation

Home

Prepare

Print/Send

Request Approval

Order

Report

Gifts

Actions

Related

Automate

Fewer options

➕ Add Free Gifts

General

Customer No.10000

ContactRobert Townes

Opportunity No.

Customer NameAdatum Corporation

No. of Archived Versions0

Responsibility Center

Address192 Market Square

Document Date4/11/2022

Assigned User ID

Address 2

Posting Date4/11/2022

StatusOpen

CityAtlanta

Order Date4/11/2022

Work Description

StateGA

Requested Delivery Date

ZIP Code31772

Promised Delivery Date

Country/Region CodeUS

External Document No.

Contact No.CT000001

Your Reference

Phone No.

Salesperson CodeJO

Mobile Phone No.

Campaign No.

Emailrobert.townes@contoso.com

Lines

Manage

Line

Order

Fewer options

Type	No.	Item Reference No.	Description	Location Code	Quantity	Qty. to Assemble to Order	Reserved Quantity	Unit of Measure Code	Unit Price Excl. Tax	Tax Area Code	Tax Group Code	Line Discount %	Line Amount Excl. Tax	Amount Including
→ Item	*		*											0

Subtotal Excl. Tax (USD)0.00

Invoice Discount %0

Total Tax (USD)0.00

Inv. Discount Amount Excl. Tax (USD)0.00

Total Excl. Tax (USD)0.00

Total Incl. Tax (USD)0.00

Invoice Details >

1M(USD)YesATLANTA, GA

Figure 4.9: Sales Order page

Codeunit definition

All of the business logic for handling the requirements for gift campaigns is defined in a Gift Management codeunit, as follows:

```
codeunit 50101 "PKT Gift Management"
{
    procedure AddGifts(var SalesHeader: Record "Sales Header")
    var
        SalesLine: Record "Sales Line";
        Handled: Boolean;
    begin
    
```

```

SalesLine.SetRange("Document Type", SalesHeader."Document Type");
SalesLine.SetRange("Document No.", SalesHeader."No.");
SalesLine.SetRange(Type, SalesLine.Type::Item);
//We exclude the generated gifts lines in order to avoid loops
SalesLine.SetFilter("Line Discount %", '<>100');
if SalesLine.FindSet() then
    repeat
        //Integration event raised
        OnBeforeFreeGiftSalesLineAdded(SalesLine, Handled);
        AddFreeGiftSalesLine(SalesHeader, SalesLine, Handled);
        //Integration Event raised
        OnAfterFreeGiftSalesLineAdded(SalesLine);
    until SalesLine.Next() = 0;
end;

local procedure AddFreeGiftSalesLine(var SalesHeader: Record "Sales
Header"; var SalesLine: Record "Sales Line"; var Handled: Boolean)
var
    GiftCampaign: Record "PKT Gift Campaign";
    Customer: Record Customer;
    SalesLineGift: Record "Sales Line";
    LineNo: Integer;
begin
    if Handled then
        exit;
    Customer.Get(SalesLine."Sell-to Customer No.");
    GiftCampaign.SetRange(CustomerCategoryCode, Customer."PKT Customer
Category Code");
    GiftCampaign.SetRange(ItemNo, SalesLine."No.");
    GiftCampaign.SetFilter(StartingDate, '<=%1', SalesHeader."Order Date");
    GiftCampaign.SetFilter(EndingDate, '>=%1', SalesHeader."Order Date");
    GiftCampaign.SetRange(Inactive, false);
    GiftCampaign.SetFilter(MinimumOrderQuantity, '<= %1', SalesLine.
Quantity);
    if GiftCampaign.FindFirst() then begin
        //Active promo found. We need to insert a new Sales Line
        LineNo := GetLastSalesDocumentLineNo(SalesHeader);
        SalesLineGift.Init();
        SalesLineGift.TransferFields(SalesLine);
        SalesLineGift."Line No." := LineNo + 10000;
    end;
end;

```

```

        SalesLineGift.Validate(Quantity, GiftCampaign.GiftQuantity);
        SalesLineGift.Validate("Line Discount %", 100);
        if SalesLineGift.Insert() then;
            end;
        end;
end;

```

Here is the procedure that calculates the last sales line number for a new document:

```

local procedure GetLastSalesDocumentLineNo(SalesHeader: Record "Sales
Header"): Integer
var
    SalesLine: Record "Sales Line";
begin
    SalesLine.SetRange("Document Type", SalesHeader."Document Type");
    SalesLine.SetRange("Document No.", SalesHeader."No.");
    if SalesLine.FindLast() then
        exit(SalesLine."Line No.")
    else
        exit(0);
    end;
end;

```

And here is the business logic that handles the gift assignment logic:

```

[EventSubscriber(ObjectType::Table, Database::"Sales Line",
'OnAfterValidateEvent', 'Quantity', false, false)]
local procedure CheckGiftEligibility(var Rec: Record "Sales Line")
var
    GiftCampaign: Record "PKT Gift Campaign";
    Customer: Record Customer;
    SalesHeader: Record "Sales Header";
    Handled: Boolean;
begin
    if (Rec.Type = Rec.Type::Item) then begin
        if (Customer.Get(Rec."Sell-to Customer No.)) then begin
            SalesHeader.Get(Rec."Document Type", Rec."Document No.");
            GiftCampaign.SetRange(CustomerCategoryCode, Customer."PKT
Customer Category Code");
            GiftCampaign.SetRange(ItemNo, Rec."No.");
            GiftCampaign.SetFilter(StartingDate, '<=%1', SalesHeader."Order
Date");
            GiftCampaign.SetFilter(EndingDate, '>=%1', SalesHeader."Order
Date");
            GiftCampaign.SetRange(Inactive, false);

```

```

        GiftCampaign.SetFilter(MinimumOrderQuantity, '> %1', Rec.
Quantity);

        if GiftCampaign.FindFirst() then begin
            //Integration event raised
            OnBeforeFreeGiftAlert(Rec, Handled);
            DoGiftCheck(Rec, GiftCampaign, Handled);
            //Integration Event raised
            OnAfterFreeGiftAlert(Rec);
        end;
    end;
end;
end;

local procedure DoGiftCheck(var SalesLine: Record "Sales Line"; var
GiftCampaign: Record "PKT Gift Campaign"; var Handled: Boolean)
var
    PacktSetup: Record "PKT Packt Setup";
    GiftAlert: Label 'Attention: there is an active promotion for item %1.
if you buy %2 you can have a gift of %3';
begin
    if Handled then
        exit;
    PacktSetup.Get();
    if (SalesLine.Quantity < GiftCampaign.MinimumOrderQuantity) and
(GiftCampaign.MinimumOrderQuantity - SalesLine.Quantity <= PacktSetup."Gift
Tolerance Qty") then
        Message(GiftAlert, SalesLine."No.", Format(GiftCampaign.
MinimumOrderQuantity), Format(GiftCampaign.GiftQuantity));
    end;
}

```

Here, we have some procedures, some event subscribers, and some event publishers. The main procedure is called AddGifts and it adds the gift lines (promotions) to the sales order passed as the argument. It raises some integration events, and the main code is handled by the AddFreeGiftSalesLine procedure.

The integration events defined in this codeunit are as follows:

```

[IntegrationEvent(true, false)]
local procedure OnBeforeFreeGiftSalesLineAdded(var SalesHeader: Record
"Sales Header"; var SalesLine: Record "Sales Line"; var Handled: Boolean)
begin
end;

```

```

[IntegrationEvent(true, false)]
local procedure OnAfterFreeGiftSalesLineAdded(var SalesHeader: Record
"Sales Header"; var SalesLine: Record "Sales Line")
begin
end;

[IntegrationEvent(true, false)]
local procedure OnBeforeFreeGiftAlert(var SalesLine: Record "Sales Line";
var Handled: Boolean)
begin
end;

[IntegrationEvent(true, false)]
local procedure OnAfterFreeGiftAlert(var SalesLine: Record "Sales Line")
begin
end;

```

Here, we've implemented the Handled pattern (to guarantee extensibility). In this way, a dependent extension can change the gift assignment logic as needed without modifying the base code of the main extension.

The Handled pattern implementation is as follows:

```

OnBeforeFreeGiftSalesLineAdded(SalesHeader, SalesLine, Handled);
AddFreeGiftSalesLine(SalesHeader, SalesLine, Handled);
OnAfterFreeGiftSalesLineAdded(SalesHeader, SalesLine);

```

In this code, we have the following:

- We have a global variable called Handled set to false.
- We raise an integration event called OnBeforeFreeGiftSalesLineAdded by passing the sales header and sales line we're working on and the Handled variable.
- We implement the business logic in a procedure called AddFreeGiftSalesLine. In this procedure, if the event is handled, we skip the standard logic:

```

if Handled then
    exit;

```

- At the end of the process, we raise an integration event called OnAfterFreeGiftSalesLineAdded.

So, why does this pattern guarantee extensibility? This is because, in a dependent extension, you can subscribe to the OnBeforeFreeGiftSalesLineAdded event, set the Handled variable to true, and implement your new business logic for adding gifts. Then, the standard business logic (AddFreeGiftSalesLine) is skipped.

After this, you can subscribe to the `OnAfterFreeGiftSalesLineAdded` event and implement other custom business logic that must be executed after the process of adding gifts. We'll see an example of a dependent extension that alters the standard business logic of our extension in the following *Writing Code for Extensibility* chapter.

In this codeunit, we've also created a procedure called `CheckGiftEligibility`, which is an event subscriber of the `OnAfterValidateEvent` event of the `Quantity` field of the `Sales Line` table. The following code shows this:

```
[EventSubscriber(ObjectType::Table, Database::"Sales Line",
'OnAfterValidateEvent', 'Quantity', false, false)]
    local procedure CheckGiftEligibility(var Rec: Record "Sales Line")
```

In this function, we handle the business logic for the alert that must be triggered when the sales operator inserts the quantity in a sales line. As you can see in the preceding code, we've implemented the `Handled` pattern again here to provide extensibility.

What happens when, from a sales order for a customer where you previously assigned a customer category that allows gifts, you trigger the **Add Free Gifts** action? Refer to the following screenshot:

The screenshot shows the 'Sales Order' form for S-ORD101005. The 'Add Free Gifts' button is highlighted with a red box. The form displays customer information for Adatum Corporation and a list of sales lines. The third line, 'ATHENS Desk', is highlighted with a red box and has a 'Line Discount %' of 100.

Type	No.	Item Reference No.	Description	Location Code	Quantity	Qty. to Assemble to Order	Reserved Quantity	Unit of Measure Code	Unit Price Excl. Tax	Tax Area Code	Tax Group Code	Line Discount %
Item	1896-S		ATHENS Desk		5			PCS	1,000.80	ATLANTA, GA	FURNITURE	
Item	1900-S		PARIS Guest Chair, black		10			PCS	192.80	ATLANTA, GA	FURNITURE	
Item	1896-S		ATHENS Desk		1			PCS	1,000.80	ATLANTA, GA	FURNITURE	100

Figure 4.10: Add Free Gifts behavior

We can see that the events are raised, the `AddGifts` function is executed, and the gift promotions (if any) are inserted in the `Sales Line` table (a new line with the **Line Discount %** field value set to 100).

We've now implemented all of the business requirements needed to manage the customer's gift campaigns. Now, let's move on to the vendor quality implementation details.

## Vendor quality implementations

To handle the vendor quality management requirements, we need to do the following:

- Define a Vendor Quality table (related to the standard Vendor table) that will contain details about the quality scores for vendor and quality-related financial data.
- Define the relevant card page and attach it to the vendor card (this will be the quality detail card for a vendor as per our requirements).
- Add a new action to the standard Vendor Card page to open Vendor Quality Card.
- Define a codeunit that handles all of the business logic related to this implementation.

In the following sections, we'll see the various object implementations in detail.

### Table definition

By using the `ttable` snippet, we define the **Vendor Quality** table as follows:

```
table 50102 "PKT Vendor Quality"
{
    Caption = 'Vendor Quality';
    DataClassification = CustomerContent;

    fields
    {
        field(1; "Vendor No."; Code[20])
        {
            Caption = 'Vendor No.';
            TableRelation = Vendor;
        }
        field(2; "Vendor Name"; Text[100])
        {
            Caption = 'Vendor Name';
            FieldClass = FlowField;
            CalcFormula = lookup(Vendor.Name where("No." = field("Vendor
No.")));
            Editable = false;
        }
        field(3; "Vendor Activity Description"; Text[250])
        {
            Caption = 'Vendor Activity Description';
        }
        field(4; ScoreItemQuality; Integer)
        {
```

```
Caption = 'Item Quality Score';
MinValue = 1;
MaxValue = 10;

trigger OnValidate()
begin
    UpdateVendorRate();
end;
}
field(5; ScoreDelivery; Integer)
{
    Caption = 'Delivery On Time Score';
    MinValue = 1;
    MaxValue = 10;

    trigger OnValidate()
    begin
        UpdateVendorRate();
    end;
}
field(6; ScorePackaging; Integer)
{
    Caption = 'Packaging Score';
    MinValue = 1;
    MaxValue = 10;

    trigger OnValidate()
    begin
        UpdateVendorRate();
    end;
}
field(7; ScorePricing; Integer)
{
    Caption = 'Pricing Score';
    MinValue = 1;
    MaxValue = 10;

    trigger OnValidate()
    begin
        UpdateVendorRate();
    end;
}
```

```
    }
    field(8; Rate; Decimal)
    {
        Caption = 'Vendor Rate';
    }
    field(10; UpdateDate; DateTime)
    {
        Caption = 'Update Date';
    }

    field(11; InvoicedYearN; Decimal)
    {
        Caption = 'Invoiced for current year (N)';
    }
    field(12; InvoicedYearN1; Decimal)
    {
        Caption = 'Invoiced for year N-1';
    }
    field(13; InvoicedYearN2; Decimal)
    {
        Caption = 'Invoiced for year N-2';
    }
    field(14; DueAmount; Decimal)
    {
        Caption = 'Due Amount';
        DataClassification = CustomerContent;
    }
    field(15; AmountNotDue; Decimal)
    {
        Caption = 'Amount to pay (not due)';
    }
}

keys
{
    key(PK; "Vendor No.")
    {
        Clustered = true;
    }
}
```

In the table's triggers, we handled some initialization of values:

```

trigger OnInsert()
begin
    UpdateDate := CurrentDateTime();
end;

trigger OnModify()
begin
    UpdateDate := CurrentDateTime();
end;

local procedure UpdateVendorRate()
var
    VendorQualityMgt: Codeunit "PKT Vendor Quality Mgt";
begin
    VendorQualityMgt.CalculateVendorRate(Rec);
end;
}

```

In this table, we have the definitions of the required score fields (rating) and the required financial fields.

For the score ratings, we handle the OnValidate trigger to dynamically update the rate calculation when the user inserts values in the fields (this is done by calling the UpdateVendorRate function defined in the table (as a class method) but implemented in the external codeunit that we'll see later).

We've also handled the table's OnInsert and OnModify triggers to save the insertion or modification date of the record (business requirements).

## Page definition

For our business requirements, we need to create a Vendor Quality Card page. We create a new page of the Card type by using the tpage snippet, as follows:

```

page 50102 "PKT Vendor Quality Card"
{
    Caption = 'Vendor Quality Card';
    SourceTable = "PKT Vendor Quality";
    PageType = Card;
    ApplicationArea = All;
    UsageCategory = Administration;
    InsertAllowed = false;
    AboutTitle = 'About Vendor Quality';
}

```

```
AboutText = '**Vendor Quality** gives you an overview on how your Vendors  
performs and how they are ranked internally in your company.';
```

```
layout  
{  
    area(Content)  
    {  
        group(General)  
        {  
            Caption = 'General';  
            field("Vendor No."; Rec."Vendor No.")  
            {  
                Editable = false;  
            }  
            field("Vendor Name"; Rec."Vendor Name")  
            {  
                Editable = false;  
            }  
            field("Vendor Activity Description"; Rec."Vendor Activity  
Description")  
            {  
            }  
            field(Rate; Rec.Rate)  
            {  
                Editable = false;  
                Style = Strong;  
            }  
            field(UpdateDate; Rec.UpdateDate)  
            {  
                Editable = false;  
            }  
        }  
        group(Scoring)  
        {  
            Caption = 'Score';  
            field(ScoreItemQuality; Rec.ScoreItemQuality)  
            {  
            }  
            field(ScoreDelivery; Rec.ScoreDelivery)  
            {  
            }  
        }  
    }  
}
```

```

    }
    field(ScorePackaging; Rec.ScorePackaging)
    {
    }
    field(ScorePricing; Rec.ScorePricing)
    {
    }
}
group(Financials)
{
    Caption = 'Financials';
    field(InvoicedYearN; Rec.InvoicedYearN)
    {
        Editable = false;
    }
    field(InvoicedYearN1; Rec.InvoicedYearN1)
    {
        Editable = false;
    }
    field(InvoicedYearN2; Rec.InvoicedYearN2)
    {
        Editable = false;
    }
    field(DueAmount; Rec.DueAmount)
    {
        Editable = false;
        Style = Attention;
    }
    field(AmountNotDue; Rec.AmountNotDue)
    {
        Editable = false;
    }
}
}

trigger OnOpenPage()
begin
    if not Rec.Insert() then;
end;

```

```

trigger OnAfterGetRecord()
var
    VendorQualityMgt: Codeunit "PKT Vendor Quality Mgt";
begin
    VendorQualityMgt.UpdateVendorQualityStatistics(Rec);
end;
}

```

This page is designed by creating different groups (FastTabs in the UI):

- **General:** Contains the general quality classification of the selected vendor, such as the name, a description of the activity, and the calculation rate
- **Scoring:** Contains the quality scores (as assigned by the company's quality manager)
- **Financials:** Contains the financial data required from the quality requirements

This page has the `InsertAllowed` property set to `true` because the record here is inserted automatically when the page is opened from Vendor Card (we handle the `OnOpenPage` trigger here) and the user can't directly insert new records from this page.

We also handle the `OnAfterGetRecord` page trigger, and from here, we call a function that refreshes the financial statistics.

## The pageextension definition

We need a pageextension object to create the action of opening the previously created Vendor Quality Card page from the standard Vendor Card page. By using the `tpageext` snippet, we create the following object:

```

pageextension 50101 "PKT VendorCardExt" extends "Vendor Card"
{
    actions
    {
        addafter("Comments")
        {
            action("PKT Quality Classification")
            {
                Caption = 'Quality Classification';
                ApplicationArea = All;
                Image = QualificationOverview;
                RunObject = Page "PKT Vendor Quality Card";
                RunPageLink = "Vendor No." = field("No.");
            }
        }
    }
}

```

```
addlast(Promoted)
{
    group(PKTQuality)
    {
        Caption = 'Vendor Quality';
        actionref(PKTQualityClassification; "PKT Quality
Classification")
        {
        }
    }
}
```

Here, we have defined a QualityClassification action, which opens the Vendor Quality Card page for the selected Vendor record (by using the RunPageLink property).

The page action appears as follows:

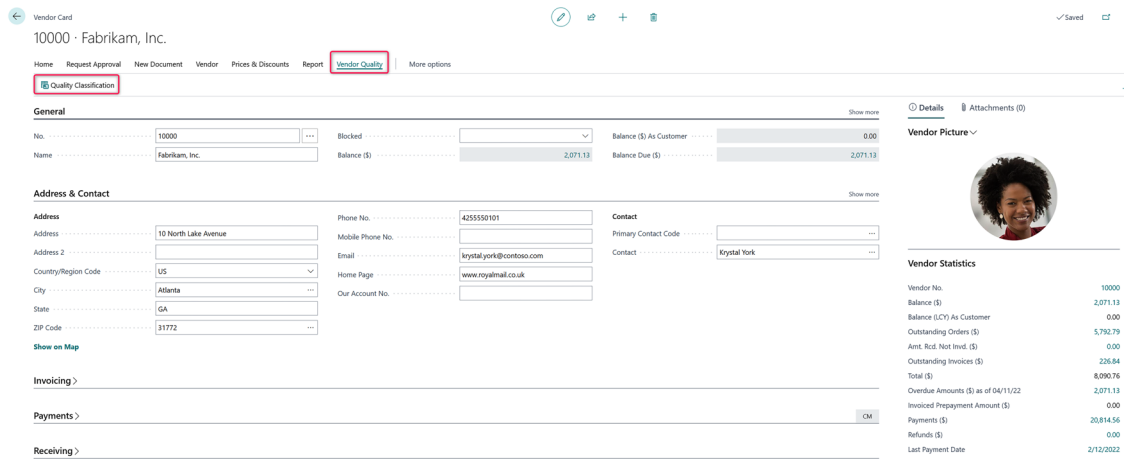


Figure 4.11: Vendor Quality Card page behavior

When triggering the action, the Vendor Quality Card page is opened and looks like this:

←

Vendor Quality Card

✓ Saved

10000

General

Vendor No. ....

10000

Vendor Rate .....

8.00

Vendor Name .....

Fabrikam, Inc.

Update Date .....

12/28/2022 3:51 PM

Vendor Activity Descri...

Supplier of goods

Score

Item Quality Score .....

8

Packaging Score .....

7

Delivery On Time Score ..

8

Pricing Score .....

9

Financials

Invoiced for current y...

4,959.90

Due Amount .....

2,071.13

Invoiced for year N-1 ...

16,672.30

Amount to pay (not d...

0.00

Invoiced for year N-2 ...

0.00

Figure 4.12: Vendor Quality Card page

When the quality manager inserts score values (manually inserted votes under the **Score** tab), the Vendor Rate value is automatically calculated. Financial statistics are automatically calculated in real time when opening the page.

## Codeunit definition

As usual, we define all of our business logic in an external codeunit called Vendor Quality Mgt using the tcodeunit snippet, defined as follows:

```
codeunit 50102 "PKT Vendor Quality Mgt"
{
    procedure CalculateVendorRate(var VendorQuality: Record "PKT Vendor
Quality")
    var
        Handled: Boolean;
    begin
        OnBeforeCalculateVendorRate(VendorQuality, Handled);
        //This is the company's criteria to assign the Vendor rate.
        VendorRateCalculation(VendorQuality, Handled);
        OnAfterCalculateVendorRate(VendorQuality);
    end;

    local procedure VendorRateCalculation(var VendorQuality: Record "PKT Vendor
Quality"; var Handled: Boolean)
    begin
        if Handled then
            exit;
        VendorQuality.Rate := (VendorQuality.ScoreDelivery + VendorQuality.
ScoreItemQuality +
            VendorQuality.ScorePackaging + VendorQuality.ScorePricing) / 4;
    end;

    procedure UpdateVendorQualityStatistics(var VendorQuality: Record "PKT
Vendor Quality")
    var
        Year: Integer;
        DW: Dialog;
        DialogMessage: Label 'Calculating Vendor statistics...';
    begin
        DW.Open(DialogMessage);
        Year := DATE2DMY(TODAY, 3);
        VendorQuality.InvoicedYearN := GetInvoicedAmount(VendorQuality."Vendor
No.", DMY2DATE(1, 1, Year), TODAY);
        VendorQuality.InvoicedYearN1 := GetInvoicedAmount(VendorQuality."Vendor
No.", DMY2DATE(1, 1, Year - 1), DMY2DATE(31, 12, Year - 1));
```

```

        VendorQuality.InvoicedYearN2 := GetInvoicedAmount(VendorQuality."Vendor
No.", DMY2DATE(1, 1, Year - 2), DMY2DATE(31, 12, Year - 2));
        VendorQuality.DueAmount := GetDueAmount(VendorQuality."Vendor No.",
TRUE);
        VendorQuality.AmountNotDue := GetDueAmount(VendorQuality."Vendor No.",
FALSE);
        DW.Close();
    end;

    local procedure GetInvoicedAmount(VendorNo: Code[20]; StartDate: Date;
EndDate: Date): Decimal
    var
        VendorLedgerEntry: Record "Vendor Ledger Entry";
        Total: Decimal;
    begin
        VendorLedgerEntry.SetRange("Vendor No.", VendorNo);
        VendorLedgerEntry.SetFilter("Document Date", '%1..%2', StartDate,
EndDate);
        VendorLedgerEntry.SetLoadFields("Purchase (LCY)");
        VendorLedgerEntry.CalcSums("Purchase (LCY)");
        exit(VendorLedgerEntry."Purchase (LCY)" * (-1));
    end;

    local procedure GetDueAmount(VendorNo: Code[20]; Due: Boolean): Decimal
    var
        VendorLedgerEntry: Record "Vendor Ledger Entry";
        Total: Decimal;
    begin
        VendorLedgerEntry.SetRange("Vendor No.", VendorNo);
        VendorLedgerEntry.SetRange(Open, TRUE);
        if Due then
            VendorLedgerEntry.SetFilter("Due Date", '< %1', TODAY)
        else
            VendorLedgerEntry.SetFilter("Due Date", '> %1', TODAY);
        VendorLedgerEntry.SETAUTOCALCFIELDS(VendorLedgerEntry."Remaining Amt.
(LCY)");
        if VendorLedgerEntry.FindSet() then
            repeat
                Total += VendorLedgerEntry."Remaining Amt. (LCY)";
            until VendorLedgerEntry.Next() = 0;

```

```

        exit(Total * (-1));
    end;

    [IntegrationEvent(true, false)]
    local procedure OnBeforeCalculateVendorRate(var VendorQuality: Record "PKT
Vendor Quality"; var Handled: Boolean)
    begin
    end;
    [IntegrationEvent(true, false)]
    local procedure OnAfterCalculateVendorRate(var VendorQuality: Record "PKT
Vendor Quality")
    begin
    end;
}

```

Here, we have defined the following functions:

- **CalculateVendorRate:** This is the function that calculates the vendor rate based on the quality scores assigned by the quality manager. We want this function to be extendable to be able to change the standard rate algorithm as needed in the future. To do that, we use the **Handled** pattern:
  - We raise an **OnBeforeCalculateVendorRate** event with the current **Vendor Quality** record and the **Handled** Boolean variable as event parameters.
  - We perform the standard rate calculation in the **VendorRateCalculation** function by checking the **Handled** parameter, and we exit from the function if we want to skip the standard calculation (by setting **Handled = true**).
  - We raise an **OnAfterCalculateVendorRate** event for handling post calculation operations or for handling a totally new calculation.
- **UpdateVendorQualityStatistics:** This function calculates the financial statistics required by the quality manager:
  - **GetInvoicedAmount:** For a given **Vendor No.** field and date period (start/end date), it calculates the invoiced amount by checking the **Vendor Ledger Entry** table (the **Purchase (LCY)** field). The returned result is -1 because we want the absolute value.
  - **GetDueAmount:** For a given **Vendor No.** field, it calculates the due amount (the **Due** parameter is set to **true**) or the amount to pay (the **Due** parameter is set to **false**) by checking the **Vendor Ledger Entry** table (the **Remaining Amt. (LCY)** field). The returned result is multiplied by -1 because we want the absolute value.

The integration events are defined as follows:

```
[IntegrationEvent(true, false)]
    local procedure OnBeforeCalculateVendorRate(var VendorQuality: Record "PKT
Vendor Quality"; var Handled: Boolean)
    begin
    end;

[IntegrationEvent(true, false)]
    local procedure OnAfterCalculateVendorRate(var VendorQuality: Record "PKT
Vendor Quality")
    begin
    end;
```

To satisfy the business requirements, we've also defined an event subscriber (the `teventsub` snippet) in the General Event Subscribers single instance codeunit for the `OnBeforeManualReleasePurchaseDoc` standard event defined in Microsoft's Release Purchase Document codeunit.

We use this event to raise an error during the order release phase if the vendor does not meet the company's rate criteria (that is, the minimum acceptable rate) defined in the extension's setup table.

The event subscriber implementation is as follows:

```
[EventSubscriber(ObjectType::Codeunit, Codeunit::"Release Purchase Document",
'OnBeforeManualReleasePurchaseDoc', '', false, false)]
    local procedure QualityCheckForReleasingPurchaseDoc(var PurchaseHeader:
Record "Purchase Header")
    var
        VendorQuality: Record "PKT Vendor Quality";
        PacktSetup: Record "PKT Packt Setup";
        ErrNoMinimumRate: Label 'Vendor %1 has a rate of %2 and it's under the
required minimum value (%3)';
    begin
        PacktSetup.Get();
        if VendorQuality.Get(PurchaseHeader."Buy-from Vendor No.") then begin
            if VendorQuality.Rate < PacktSetup."Minimum Accepted Vendor Rate"
then
                Error(ErrNoMinimumRate, PurchaseHeader."Buy-from Vendor No.",
Format(VendorQuality.Rate), Format(PacktSetup."Minimum Accepted
Vendor Rate"));
            end;
        end;
```

All of the customer's business requirements are now handled by our extension.

In the next section, we'll see how to enhance the customer's user experience by promoting actions and by creating customized page views in Dynamics 365 Business Central.

## Promoting actions

In Dynamics 365 Business Central, you can promote actions in the modern action bar on a page. This helps users find the most relevant action for their business processes.

Promoting actions is done in a specific section of the AL code in the page definition and contains a reference to the action.

To define promoted actions, you need to specify an area (called Promoted) in the actions section of a page or page extension. In the Promoted section, you can specify one or more actionref sections. An actionref is an object type that references an action defined on the page, and by adding it in the promoted area section, it's promoted in the user interface.

As an example, in the **Customer Category List** page, we have defined the following actions:

```
actions
{
    area(processing)
    {
        action("Create Default Category")
        {
            Image = CreateForm;
            ApplicationArea = All;
            ToolTip = 'Create default category';
            Caption = 'Create default category';

            trigger OnAction();
            var
                CustManagement: Codeunit "PKT Customer Category Mgt";
            begin
                CustManagement.CreateDefaultCategory();
            end;
        }
    }
    area(Promoted)
    {
        group(PKTCustomerCategory)
        {
            Caption = 'Customer Category';
            actionref(CreateDefaultCategory; "Create Default Category")
            {
            }
        }
    }
}
```

```

    }
  }
}

```

In the Promoted area we reference (via the `actionref` object) the action called `Create Default Category` in order to have it promoted on the page.

In the `pageextension` object of the `Customer List` page, we have defined the following action:

```

actions
{
  addlast(Processing)
  {
    action("PKT Assign Default Category")
    {
      Image = ChangeCustomer;
      ApplicationArea = All;
      Caption = 'Assign Default Category to all Customers';
      ToolTip = 'Assigns the Default Category to all Customers';

      trigger OnAction();
      var
        CustomerCategoryMgt: Codeunit "PKT Customer Category Mgt";
      begin
        CustomerCategoryMgt.AssignDefaultCategory();
      end;
    }
  }
  addlast(Promoted)
  {
    group(PKTCustomerCategory)
    {
      Caption = 'Customer Category';
      actionref(PKTAssigningDefaultCategory; "PKT Assign Default
Category")
      {
      }
    }
  }
}

```

Here, we added a new action to the existing standard page called `Assign Default Category` (added in the standard `Processing` action group). Then we promote it by adding it to the existing `Promoted` action group by adding a reference to the action (using `actionref`).

Please note that an `actionref` object can only be defined in the `area(Promoted)` section. You can either create groups in `area(Promoted)` for the `actionref` references, or you can add `actionref` sections directly. An `actionref` inherits the properties of the referenced action.

## Creating page views

In Dynamics 365 Business Central, you can create customized views for your list pages. These customized views can be used in a dedicated section of the Dynamics 365 Business Central user interface to immediately apply filters to the list.

You can create a view definition in a page object by using the `tview` snippet. In the previously created `Gift Campaign List` page, we defined the following view objects:

```
views
{
    view(ActiveCampaigns)
    {
        Caption = 'Active Gift Campaigns';
        Filters = where(Inactive = const(false));
    }
    view(InactiveCampaigns)
    {
        Caption = 'Inactive Gift Campaigns';
        Filters = where(Inactive = const(true));
    }
}
```

The first view (called `ActiveCampaigns`) shows all of the active gift campaigns (the `Inactive` field is set to `false`), while the second view (called `InactiveCampaigns`) shows all of the inactive gift campaigns (the `Inactive` field is set to `true`).

These views in the Dynamics 365 Business Central user interface look like this:

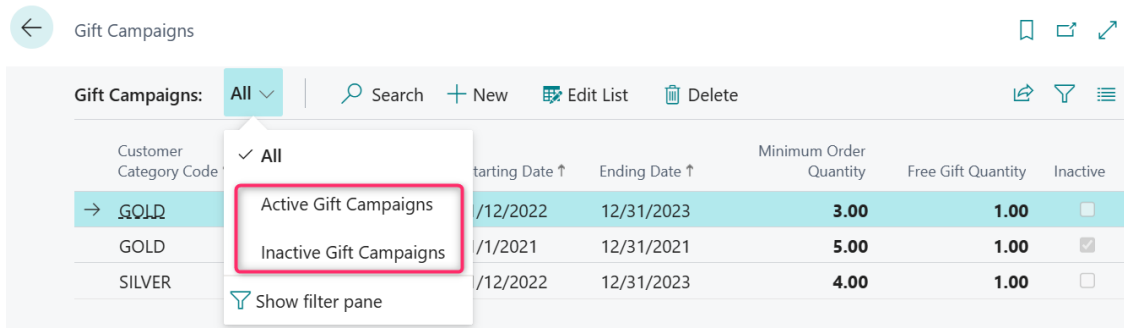


Figure 4.13: Campaign views

If you select the **Active Gift Campaigns** view, the list is filtered accordingly (Inactive is set to false):

← Gift Campaigns

Gift Campaigns:

Active Gift Campaigns ▾

Search + New Edit List Delete

Customer Category Code ↑

Item No. ↑

Starting Date ↑

Ending Date ↑

Minimum Order Quantity

Free Gift Quantity

Inactive ▾

→ GOLD	: 1896-S	1/12/2022	12/31/2023	3.00	1.00	<input type="checkbox"/>
SILVER	1896-S	1/12/2022	12/31/2023	4.00	1.00	<input type="checkbox"/>

Figure 4.14: Active campaign filter

If you select the **Inactive Gift Campaigns** view, the list is automatically filtered by Inactive being set to true:

← Gift Campaigns

Gift Campaigns:

Inactive Gift Campaigns ▾

Search + New Edit List Delete

Customer Category Code ↑

Item No. ↑

Starting Date ↑

Ending Date ↑

Minimum Order Quantity

Free Gift Quantity

Inactive ▾

→ GOLD	: 1806-S	1/1/2021	12/31/2021	5.00	1.00	<input checked="" type="checkbox"/>
--------	----------	----------	------------	------	------	-------------------------------------

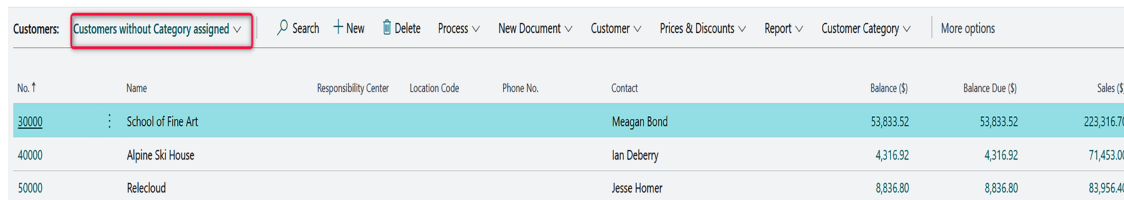
Figure 4.15: Inactive campaign filter

We have also added a view to the Customer List page to show all of the customers who do not have an associated category.

The view definition in the Customer List pageextension object is defined as follows:

```
views
{
    addlast
    {
        view(CustomersWithoutCategory)
        {
            Caption = 'Customers without Category assigned';
            Filters = where("PKT Customer Category Code" = filter(''));
        }
    }
}
```

We have placed this newly created view as the last of the available views for the page:



No. ↑	Name	Responsibility Center	Location Code	Phone No.	Contact	Balance (\$)	Balance Due (\$)	Sales (\$)
30000	School of Fine Art				Meagan Bond	53,833.52	53,833.52	223,316.70
40000	Alpine Ski House				Ian Deberry	4,316.92	4,316.92	71,453.00
50000	Relecloud				Jesse Homer	8,836.80	8,836.80	83,956.40

Figure 4.16: Selecting the new view

As shown in the preceding screenshot, this view appears in the user interface of the application and, when selected, it automatically filters all of the customers without an associated category (so Customer Category = Blank). In this way, our customers can immediately filter records by selecting a pre-defined view (just a click) and without reinserting the required filters.

After the development phase of our extensions, once everything is tested, we can move on to the extension installation process and see how installing and upgrading codeunits is done.

## Installing and upgrading codeunits

When you create an extension, you could encounter the need to check some conditions for the installation to be successful, or you could need to initiate some setup tables or pre-populate other tables. To do this, you need to create an Install codeunit.

The extension's install logic must be written in a codeunit with the SubType property set to Install. This logic is triggered when the following is true:

- You're installing the extension for the first time.
- You have uninstalled the extension and are reinstalling it again.

An Install codeunit supports the following system triggers:

- `OnInstallAppPerCompany()`: The code inside this trigger runs once for each company in the Dynamics 365 Business Central database.
- `OnInstallAppPerDatabase()`: The code inside this trigger runs once for the entire install process.

The same logic occurs when upgrading the extension. If you need to create a new version of your extension (the version number in the `app.json` file must be greater than the previous version number) and this version involves data modifications from previous versions, you need to create an Upgrade codeunit.

The extension's upgrade logic must be written in a codeunit with the SubType property set to Upgrade.

An Upgrade codeunit supports the following system triggers:

- `OnCheckPreconditionsPerCompany()`: The code inside this trigger is used to check the pre-conditions for the upgrade process. This code runs once for each company in the database.

- `OnCheckPreconditionsPerDatabase()`: The code inside this trigger is used to check the preconditions for the upgrade process. This code runs once for the entire upgrade process.
- `OnUpgradePerCompany()`: The code inside this trigger contains the upgrade logic. This code runs once for each company in the database.
- `OnUpgradePerDatabase()`: The code inside this trigger contains the upgrade logic. This code runs once for the entire upgrade process.
- `OnValidateUpgradePerCompany()`: The code inside this trigger is used to check the results of the upgrade process. This code runs once for each company in the database.
- `OnValidateUpgradePerDatabase()`: The code inside this trigger is used to check the results of the upgrade process. This code runs once for the entire upgrade process.

For our extension, we have created an `Install` codeunit as follows:

```
codeunit 50105 "PKT Packt Install"
{
    Subtype = Install;
    trigger OnInstallAppPerCompany();
    var
        CustomerCategory: Record "PKT Customer Category";
        PacktSetup: Record "PKT Packt Setup";
    begin
        if CustomerCategory.IsEmpty() then
            InsertDefaultCustomerCategory();
        if PacktSetup.IsEmpty() then
            InsertDefaultSetup();
    end;

    // Insert the GOLD, SILVER, BRONZE reward Levels
    local procedure InsertDefaultCustomerCategory();
    begin
        InsertCustomerCategory('DEFAULT', 'Default', true);
        InsertCustomerCategory('GOLD', 'Gold customers', false);
        InsertCustomerCategory('SILVER', 'Silver customers', false);
        InsertCustomerCategory('BRONZE', 'Bronze customers', false);
    end;

    // Create and insert a Customer Category record
    local procedure InsertCustomerCategory(ID: Code[20]; Description:
    Text[550]; Default: Boolean);
    var
        CustomerCategory: Record "PKT Customer Category";
    begin
```

```

    CustomerCategory.Init();
    CustomerCategory.Code := ID;
    CustomerCategory.Description := Description;
    CustomerCategory.Default := Default;
    CustomerCategory.Insert();
end;

local procedure InsertDefaultSetup()
var
    PacktSetup: Record "PKT Packt Setup";
begin
    PacktSetup.Init();
    PacktSetup."Minimum Accepted Vendor Rate" := 6;
    PacktSetup."Gift Tolerance Qty" := 2;
    PacktSetup.Insert();
end;
}

```

Here, we initialize the Customer Category table and the extension's setup table (Packt Extension Setup table) with default data if no data is present.

An Upgrade codeunit is useful (and required) when you need to handle data upgrades between different versions of your extension or you need to handle object obsolescence (data structure changes).

The common scenario of Upgrade codeunit use is moving data between tables or fields. If you have, for example, two tables (here, called FromTable and ToTable) and you want to move fields between the two tables during the upgrade process, you can do something like the following:

```

codeunit 50104 "PKT Packt Upgrade"
{
    Subtype = Upgrade;

    trigger OnUpgradePerCompany()
    begin
        MoveField();
    end;

    local procedure MoveField()
    var
        from: Record FromTable;
        to: Record ToTable;
    begin
        if from.FindSet() then
            repeat

```

```

        to.Get(from.RecordId);
        to.IntField := from.IntField;
        to.CodeField := from.CodeField;
        to.Modify();
    until from.Next() = 0
end;
}

```

If you need to move records between two tables, you can do something like the following:

```

codeunit 50104 "PKT Packt Upgrade"
{
    Subtype = Upgrade;

    trigger OnUpgradePerCompany()
    begin
        MoveRecords();
    end;

    local procedure MoveRecords()
    var
        from: Record FromTable;
        to: Record ToTable;
    begin
        if from.FindSet() then
            repeat
                to.SmallCodeField := from.SmallCodeField;
                to.IntField := from.IntField;
                to.id := from.id;
                to.Insert();
            until from.Next() = 0
        end;
    }
}

```

But what happens if you have millions of records to upgrade or move? The above code works row by row (record-level) and your upgrade code could become very slow with a huge amount of records to update. This can affect the upgrade performance and (as a consequence) also the user experience.

To support the fastest-possible data transfers during the extension upgrade process, Microsoft has created the `DataTransfer` data type. `DataTransfer` is an AL data type that supports the bulk transferring of data between SQL-based tables. It produces SQL code that operates on sets and this behavior improves the performance when moving data during upgrades.

You can rewrite the previously described procedure to move fields between tables by using the `DataTransfer` data type as follows:

```
local procedure MoveField()
var
    dt : DataTransfer;
    from: Record FromTable;
    to : Record ToTable;
begin
    dt.SetTables(Database::FromTable, Database::ToTable);
    dt.AddFieldValue(from.FieldNo("SmallCodeField"), to.FieldNo("SmallCodeField"));
    dt.AddFieldValue(from.FieldNo("IntField"), to.FieldNo("IntField"));
    dt.AddJoinCondition(from.FieldNo("id"), to.FieldNo("id"));
    dt.CopyFields();
end;
```

The same `if` you want to use `DataTransfer` to rewrite the previously described procedure to move records between tables:

```
local procedure MoveRecords()
var
    dt : DataTransfer;
    to : Record ToTable;
begin
    dt.SetTables(Database::FromTable, Database::ToTable);
    dt.AddFieldValue(2, to.FieldNo("SmallCodeField"));
    dt.AddFieldValue(3, to.FieldNo("IntField"));
    dt.AddFieldValue(1, to.FieldNo("id"));
    dt.CopyRows();
end;
```

This new code uses `DataTransfer` to map fields between the source and destination table (using the `SetTables` method), then uses the `CopyRows` method to move records in bulk.

By using the `DataTransfer` data type, you will have much faster upgrades (> 90x faster than using pure AL code). For a benchmark, I suggest checking this link:

<https://demiliani.com/2022/12/01/dynamics-365-business-central-datatransfer-on-upgrades-and-performances/>

Please remember the following limitations on the `DataTransfer` object:

- The `DataTransfer` object can only be used in upgrade code. It will throw a runtime error if used outside of upgrade codeunits.
- The `DataTransfer` object can't be used on the following tables:
  - Non-SQL tables
  - System tables
  - Virtual tables
  - Audited tables as the destination
  - Obsolete tables as the destination



More information about the `Install` and `Upgrade` codeunits can be found at the following links:

<https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-extension-install-code> and <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-upgrading-extensions>.

When published from Visual Studio Code on Dynamics 365 Business Central, our extension appears as installed on the Installed Extensions page:

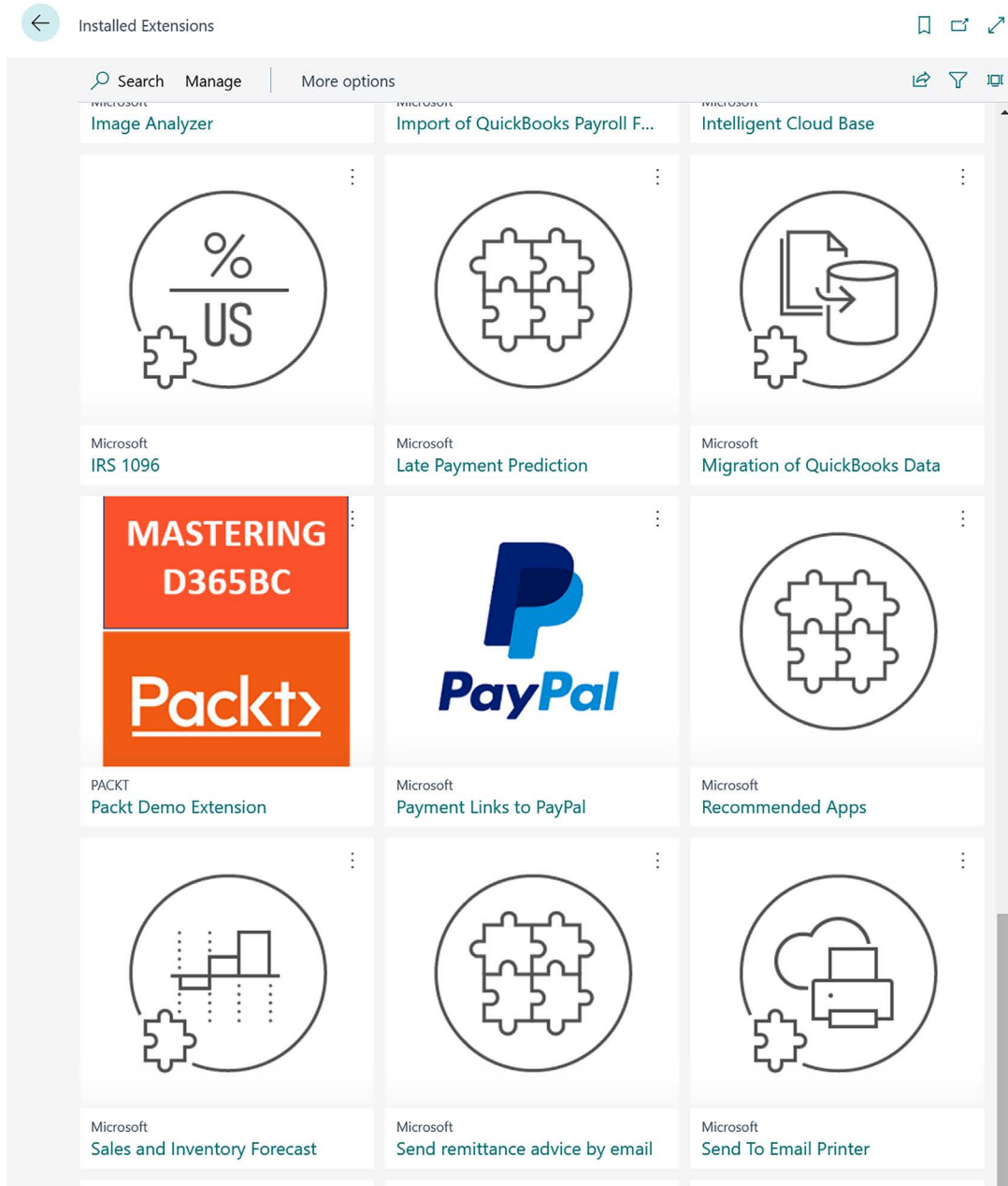


Figure 4.17: Installed Extensions page

## Defining permission sets in AL

An extension needs to contain at least a permission set. A permission set object in Dynamics 365 Business Central describes permissions on objects that you can then assign to your users for role-based access to the application.

You can define a permission set in AL by using the `tpermissionset` snippet:

```
permissionset Id MyPermissionSet
{
    Assignable = true;
    IncludedPermissionSets = SomePermissionSet;
    Permissions =
        ObjectType ObjectName = ObjectPermissions;
}
```

Here is the permission set that we have defined in our previously created extension:

```
permissionset 50100 "PKT Packt Permission"
{
    Caption = 'PACKT Permissions';
    Assignable = true;
    Permissions = tabledata "PKT Customer Category" = RIMD,
        tabledata "PKT Gift Campaign" = RIMD,
        tabledata "PKT Packt Setup" = RIMD,
        tabledata "PKT Vendor Quality" = RIMD,
        table "PKT Customer Category" = X,
        table "PKT Gift Campaign" = X,
        table "PKT Packt Setup" = X,
        table "PKT Vendor Quality" = X,
        report "Item Ledger Entry Analysis" = X,
        codeunit "PKT Customer Category Mgt" = X,
        codeunit "PKT Gift Management" = X,
        codeunit "PKT Vendor Quality Mgt" = X,
        page "PKT Customer Category Card" = X,
        page "PKT Customer Category List" = X,
        page "PKT Gift Campaign List" = X,
        page "PKT Packt Setup" = X,
        page "PKT Vendor Quality Card" = X;
}
```

The AL language extension has a new command called `al.generatePermissionSetForExtensionObjects` (that you can execute from the Visual Studio Code Command Palette) that permits you to generate or update permission sets for the active project quickly:

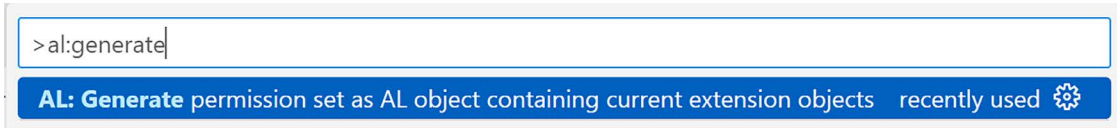


Figure 4.18: AL: Generate shown in the Command Palette

A permissionset object can be assignable to a user or not. The Assignable property defines this behavior. Assignable permission sets are permissions that an admin can assign to users in Business Central using the **Permission Sets** page. Permission sets where the Assignable property is set to false cannot be assigned to users via the user interface but can be used to compose other permission sets.

Permission sets can also include or exclude other permission sets by using the following properties:

- **IncludedPermissionSets:** This defines a list of other permission sets included in this permission set. The property is available on permission sets and permission set extension objects.
- **ExcludedPermissionSets:** This defines a list of other permission sets that are excluded from this permission set. The property is not supported on permission set extension objects.

The following diagram shows the composition of a permission set:

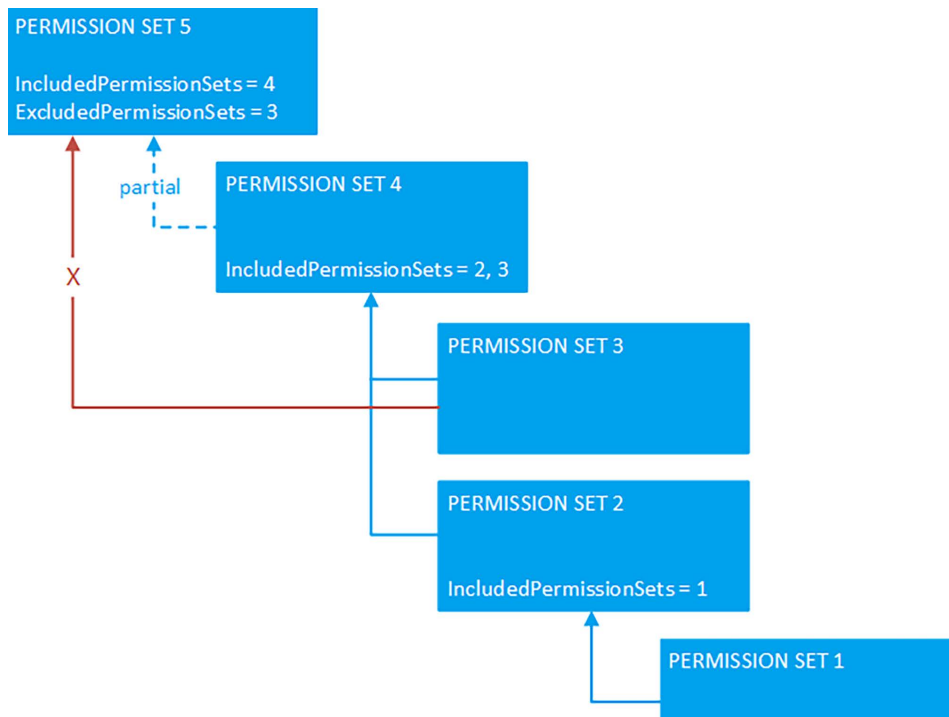


Figure 4.19: Permission set composition

Let's consider the following example of a composable permission set. We define a base permission set for sales agents as follows:

```
permissionset 50101 "PKT Sales Agent"
{
    Assignable = true;
    Caption = 'Sales Agent Base';

    Permissions =
        tabledata Customer = RIMD,
        tabledata "Payment Terms" = RMD,
        tabledata Currency = RM,
        tabledata "Sales Header" = RIM,
        tabledata "Sales Line" = RIMD;
}
```

Then we define an **Advanced Sales Agent** permission set that includes the previously defined base permission set with the addition of the possibility to edit customers and vendors and add custom permissions on specific objects:

```
permissionset 50102 "PKT Sales Agent Adv"
{
    Assignable = true;
    Caption = 'Advanced Sales Agent';
    IncludedPermissionSets = "PKT Sales Agent", "D365 CUSTOMER, EDIT", "D365
VENDOR, EDIT";

    Permissions =
        tabledata "Purchase Header" = RIM,
        tabledata "Purchase Line" = RIMD,
        codeunit AccSchedManagement = X;
}
```

Then we define another permission set called **Junior Sales Agent** than contains the advanced permissions but not the possibility to edit vendors. We do this as follows:

```
permissionset 50103 "PKT Sales Agent Jnr"
{
    Assignable = true;
    Caption = 'Junior Sales Agent';
    IncludedPermissionSets = "PKT Sales Agent Adv";
    ExcludedPermissionSets = "D365 VENDOR, EDIT";
}
```

We have now learned how to handle the install and upgrade operations required when publishing an extension in Dynamics 365 Business Central.

In the next chapter, we'll see how to write code for extensibility. We'll explore the concept of dependent extensions and learn how to use a dependency to customize our previously deployed application.

## Summary

In this chapter, we saw the implementation of a real-world extension for Dynamics 365 Business Central. We defined the backend of our solution (tables) and we created the pages (the user interface) and the required business logic (codeunits and events) according to the initial needs of the business. We saw how to make our code extensible by using the **Handled** pattern and how to create installation and upgrade code. We also learned how to create extensions with objects and events, how to use coding rules, and how to create customizations without modifying the base code of our application.

In the next chapter, we'll explore various possibilities to write extendable code and we'll see how we can write a dependent extension to customize the behavior of an existing extension.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>



# 5

## Writing Code for Extensibility

In the previous chapter, we created a full extension for Dynamics 365 Business Central to solve a real business case, by adding new entities and functionalities and customizing existing entities. We also used events to create code that can be extended in the future. However, more can be done in terms of the extensibility of our Dynamics 365 Business Central solution.

Extensibility refers to the ability of a solution to allow its capabilities to be extended without rewriting its code. As we will see, this is highly important when selling extensions to customers. If the code's basic structure is changed for each customer, we will create repeated code and increase the amount of code that needs maintenance. By considering the extensibility of our code, we can approach customer requirements differently and preserve our base extension.

This chapter will cover the following topics:

- Why we need extendible code in Dynamics 365 Business Central
- Using patterns for extensibility
- Using interfaces in AL
- Understanding dependent extensions

### **Why do we need extendible code?**

When writing extensions, we always need to think about the extensibility of our code base.

In the previous chapter, we developed an extension that adds lots of functionality to Dynamics 365 Business Central. Now, let's imagine that we want to start selling this extension to different customers. Customers start using our extension, and after a while, they start asking for customizations.

What we want to *absolutely avoid* is represented in the following schema:

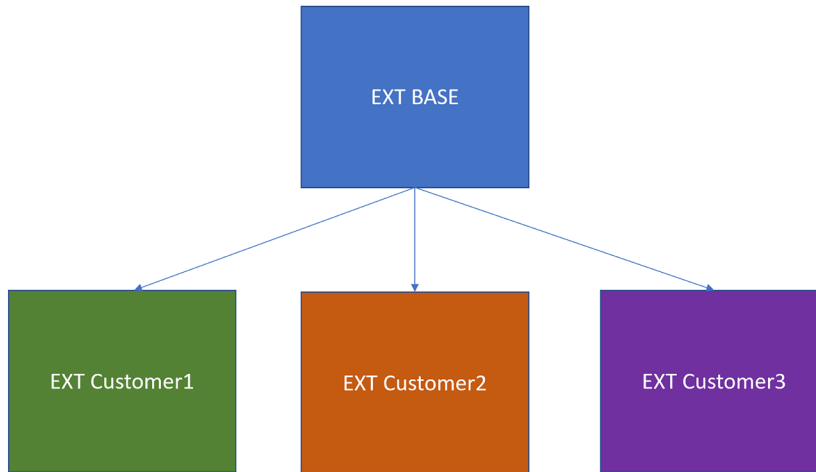


Figure 5.1: Flawed extension customer model

Here (like in the old Dynamics NAV approach), we will directly modify our base application code (our previously created extension) for the different customers. This results in a few limitations:

- Our code is replicated and modified for  $N$  customers
- We need to maintain and support  $N$  code bases
- We cannot publish our application on Microsoft's AppSource marketplace (maybe for reselling it to global Dynamics 365 Business Central customers)

To avoid these issues, we want to achieve the scenario represented in the following schema:

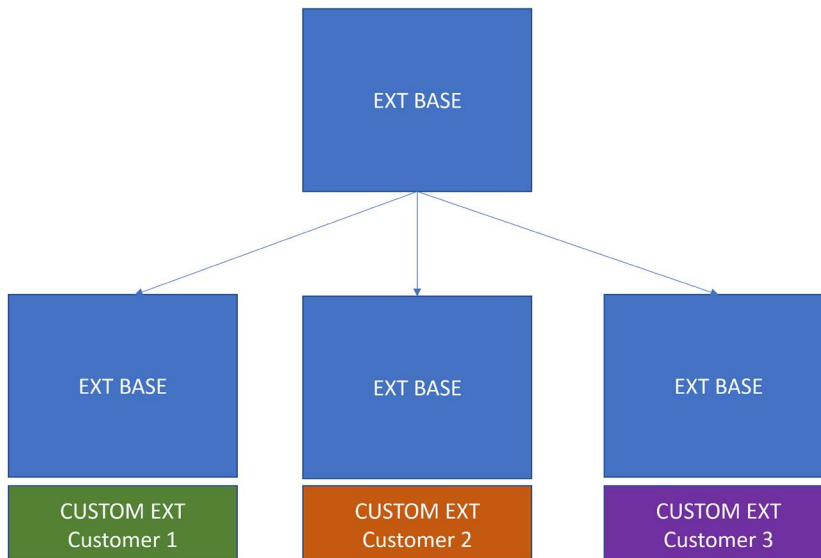


Figure 5.2: Improved extension customer model

Here, for each of the different customers that we have, we deploy the same base extension, and then we create a new customer-specific extension on top of it to handle the customizations for the specific customer. This will permit us to keep base functionalities and customer-specific functionalities completely separated.

This is the required scenario if we want to publish an application on Microsoft's AppSource marketplace, and this is the best practice to follow in every case. To do that, we need to:

- Write a code that can be extended
- Develop customer-specific features as a separate extension that has a dependency on our base extension

## Business scenario

Let's consider the following business scenario.

In our previously developed application, we want to add a new feature to calculate shipment commissions for sales orders. The requirements are as follows:

- To calculate commissions (charges) for a sales order, we need to consider only the lines where *Type = Item*.
- For all these lines, the calculation must be done in the following way:
  - If the line quantity is  $< 10$ , we need to add 1.5 \$ to the charges
  - If the line quantity is  $\geq 10$ , we need to add 5 \$ to the charges
- The charges must be added to the sales order as a new line where *Type = Charge (Item)*.
- The new charge line must be calculated and added automatically during the release process of the sales order.

This will be our base solution (the solution that we sell to all our customers).

When implementing this solution, the following must apply:

- The charge calculation process must be customizable.
- The charge calculation process can be replaced by a custom calculation for every customer.

How can we create such a process and achieve our goals? In the next section, we'll see two possible approaches, one by using events and extensibility patterns and one by using interfaces.

## Events and the "Handled" pattern

When thinking about writing code for extensibility, the first thing to keep in mind is: use events!

Using events is the first step for extensibility. Events can help you to reduce or eliminate the need to modify a base code for creating customizations.

Events allow you to create solutions that can react to actions or behaviors that occur in another solution, enabling you to create separate customized functionality from the application business logic.

Events in the AL language can be:

- Business events
- Integration events
- Internal events (subscribed only within the same module)
- Global events (predefined system events)
- Trigger events

We have previously explained and used events during the development of our extension. Here, we'll see how we can use events in a quite famous extensibility pattern in AL: the *Handled* pattern.

The *Handled* pattern is a common loose-coupling pattern that allows you to build specific points within your application where functionality can be bypassed.

In this extensibility pattern, the base application (which defines the standard business logic) will do the following:

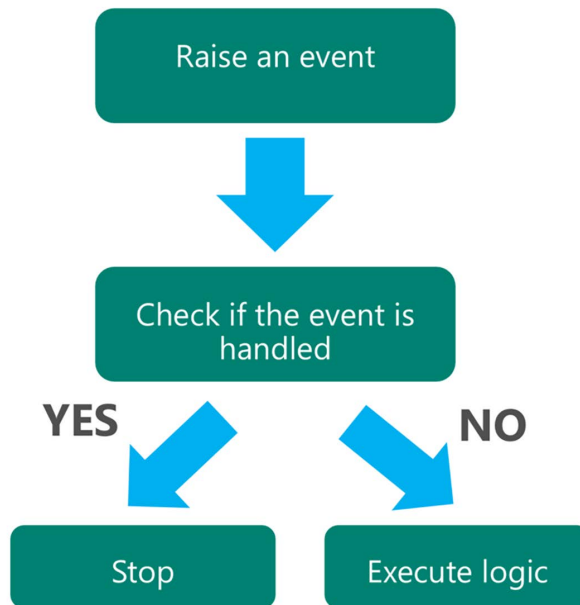


Figure 5.3: The *Handled* pattern

The base application (in this scenario, our main extension) defines a business logic for the charge calculation and publishes an event. Then, it checks if someone has “handled” that event, and if the answer is positive, the execution of the business process is stopped. If the answer is negative, the standard business process is executed.

If another extension needs to customize the standard business process, it must subscribe to the event raised by the main application and mark it as “handled,” and then it can execute its custom business process. The schema is the following:

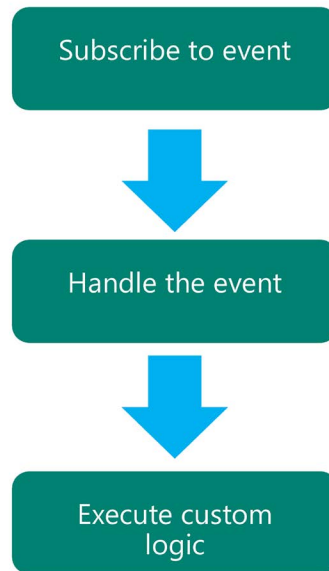


Figure 5.4: Event handling process

To define our charges calculation process and to apply this pattern to the process definition in the AL language, we have added a new codeunit object called `PKT Shipment Commission Mgt` to our previously developed extension.

The code is in the GitHub repo here:

<https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition>

The codeunit is defined as follows:

```
codeunit 50104 "PKT Shipment Commission Mgt"
{
    procedure GetShipmentCommission(SalesHeader: Record "Sales Header"; var
    total: Decimal)
    var
        SalesLine: Record "Sales Line";
        total: Decimal;
        Handled, HandledLine : Boolean;
    begin
        OnBeforeGetShipmentCommission(SalesHeader, Handled, total);
        if Handled then
            exit;
        SalesLine.SetRange("Document Type", SalesHeader."Document Type");
```

```

SalesLine.SetRange("Document No.", SalesHeader."No.");
SalesLine.SetRange(Type, SalesLine.Type::Item);
if SalesLine.FindSet() then
    repeat
    begin
        OnBeforeCalculateShipmentCommissionLine(SalesLine, total,
HandledLine);
        if not HandledLine then begin
            if SalesLine.Quantity < 10 then
                total += 1.5
            else
                total += 5;
            end;
            OnAfterCalculateShipmentCommissionLine(SalesLine, total);
        end
    until SalesLine.Next() = 0;
    OnAfterGetShipmentCommission(SalesHeader, total);
    exit(total);
end;

```

Here, we have defined a standard business process to calculate the shipment commissions, and we have also added events in order to be able to modify (extend) our business process from other extensions.

The procedure that adds the item charges to the sales order is defined as follows:

```

local procedure AddItemCharge(SalesHeader: Record "Sales Header";
totalCharge: Decimal)
var
    SalesLine: Record "Sales Line";
    PacktSetup: Record "PKT Packt Setup";
    MissingDefaultChargeItem: Label 'Missing Default Charge (Item) in Packt
Setup.';
begin
    PacktSetup.Get();
    if PacktSetup."Default Charge (Item)" = '' then
        Error(MissingDefaultChargeItem);
    SalesLine.Init();
    SalesLine."Document Type" := SalesHeader."Document Type";
    SalesLine."Document No." := SalesHeader."No.";
    SalesLine."Line No." := GetNextLineNo(SalesHeader);
    SalesLine.Validate(Type, SalesLine.type::"Charge (Item)");
    SalesLine.Validate("No.", PacktSetup."Default Charge (Item)");
    SalesLine.Validate(Quantity, 1);

```

```

        SalesLine.Validate("Unit Price", totalCharge);
        SalesLine.Insert(true);
    end;

    local procedure GetNextLineNo(SalesHeader: Record "Sales Header"): Integer
    var
        SalesLine: Record "Sales Line";
        LineNo: Integer;
    begin
        SalesLine.SetRange("Document Type", SalesHeader."Document Type");
        SalesLine.SetRange("Document No.", SalesHeader."No.");
        if SalesLine.FindLast() then
            LineNo := SalesLine."Line No." + 10000
        else
            LineNo := 10000;
        exit(LineNo);
    end;

```

These are the integration events that we use in our business process:

```

[IntegrationEvent(true, false)]
local procedure OnBeforeGetShipmentCommission(var SalesHeader: Record
"Sales Header"; var Handled: Boolean; var total: Decimal)
begin
end;

[IntegrationEvent(true, false)]
local procedure OnAfterGetShipmentCommission(var SalesHeader: Record "Sales
Header"; var Total: Decimal)
begin
end;

[IntegrationEvent(true, false)]
local procedure OnBeforeCalculateShipmentCommissionLine(var SalesLine:
Record "Sales Line"; var Total: Decimal; var HandledLine: Boolean)
begin
end;

[IntegrationEvent(true, false)]
local procedure OnAfterCalculateShipmentCommissionLine(var SalesLine:
Record "Sales Line"; var Total: Decimal)
begin
end;
}

```

In this code, we have a procedure called `GetShipmentCommission` that handles the standard calculation of our charges according to the previously described business requirements.

In the procedure's code, we have applied the `Handled` pattern in the `OnBeforeGetShipmentCommission` and `OnBeforeCalculateShipmentCommissionLine` events:

```
1 reference
procedure GetShipmentCommission(SalesHeader: Record "Sales Header"; var total: Decimal)
var
    SalesLine: Record "Sales Line";
    Handled, HandledLine : Boolean;
begin
    OnBeforeGetShipmentCommission(SalesHeader, Handled, total);
    if Handled then
        exit;
    SalesLine.SetRange("Document Type", SalesHeader."Document Type");
    SalesLine.SetRange("Document No.", SalesHeader."No.");
    SalesLine.SetRange(Type, SalesLine.Type::Item);
    if SalesLine.FindSet() then
        repeat
            begin
                OnBeforeCalculateShipmentCommissionLine(SalesLine, total, HandledLine);
                if not HandledLine then begin
                    if SalesLine.Quantity < 10 then
                        total += 1.5
                    else
                        total += 5;
                    end;
                OnAfterCalculateShipmentCommissionLine(SalesLine, total);
            end
        until SalesLine.Next() = 0;
    OnAfterGetShipmentCommission(SalesHeader, total);
end;
```

Figure 5.5: Procedure code with the `Handled` pattern

As you can see in the first box in the code, if someone subscribes to the `OnBeforeGetShipmentCommission` event and sets the `Handled` variable to `TRUE`, the standard code in the next lines is skipped, and then a custom code can be injected. The same applies to the `OnBeforeCalculateShipmentCommissionLine` event: if someone subscribes to that event and sets the `HandledLine` parameter to `TRUE`, the calculation for a sales line is skipped, and a custom code for calculating charges for a line can be injected.

The `Handled` pattern here permits us to have a business process that can be extended (customized) and permits us to replace the execution of a standard code in favor of the execution of a custom code.

The result of the `GetShipmentCommission` calculation (the total decimal value) is then passed to the `AddItemCharge` procedure, which:

- Reads the extension's setup to retrieve a custom field called `Default Charge (Item)`, used to set up the default item charge value.
- Creates the charge line in the sales order document

As per business requirements, we want to add the calculated item charge line automatically during the release process of the sales order. To do that, we subscribe to the `OnBeforeReleaseSalesDoc` event published by the standard Release Sales Document codeunit:

```
[EventSubscriber(ObjectType::Codeunit, Codeunit::"Release Sales Document",
'OnBeforeReleaseSalesDoc', '', false, false)]
    local procedure AssignShipmentCommission(var SalesHeader: Record "Sales
Header")
    var
        total: Decimal;
    begin
        GetShipmentCommission(SalesHeader, total);
        AddItemCharge(SalesHeader, total);
    end;
```

Let's now test the newly created feature. We have the following sales order document, with three lines of Type = Item and where Status is Open:

Sales Order

S-ORD101005 · Adatum Corporation

Home

Prepare

Print/Send

Request Approval

Order

Report

Gifts

Actions

Related

Automate

Fewer options

Post

Release

Create Warehouse Shipment

Create Inventory Put-away/Pick

Archive Document

General

Show less

Customer No. 10000

Contact Robert Townes

Opportunity No.

Customer Name Adatum Corporation

No. of Archived Versions 0

Responsibility Center

Sell-to

Document Date 4/11/2022

Assigned User ID

Address 192 Market Square

Posting Date 4/11/2022

Status Open

Address 2

Order Date 4/11/2022

Work Description

City Atlanta

Due Date 5/11/2022

State GA

Requested Delivery Date

ZIP Code 31772

Promised Delivery Date

Country/Region Code US

External Document No.

Contact No. CT000001

Your Reference

Phone No.

Salesperson Code JO

Mobile Phone No.

Campaign No.

Email robert.townes@contoso.com

Lines

Manage

Line

Order

Fewer options

Type	No.	Item Reference No.	Description	Location Code	Quantity	Qty. to Assemble to Order	Reserved Quantity	Unit of Measure Code	Unit Price Excl. Tax	Tax Area Code	Tax Group Code	Line Discount %	Line Amount Excl. Tax	Amount Including
→ Item	1896-S		ATHENS Desk		5	-	-	PCS	1,000.80	ATLANTA, GA	FURNITURE		5,004.00	5,304
Item	1900-S		PARIS Guest Chair, black		10	-	-	PCS	192.80	ATLANTA, GA	FURNITURE		1,928.00	2,043
Item	1896-S		ATHENS Desk		1	-	-	PCS	1,000.80	ATLANTA, GA	FURNITURE	100		0

Figure 5.6: Sales order with new feature

Now, if we release the order, a new line (where Type = Charge (Item)) is created with the calculated charges:

Sales Order

101005 - Adatum Corporation

Home Prepare Print/Send Request Approval Order Gifts More options

Post... Release... Create Warehouse Shipment Create Inventory Put-away/Pick... Archive Document

**General** Show more

Customer No. 10000 Posting Date 4/2/2023 External Document No.   
 Customer Name Adatum Corporation VAT Date 4/2/2023 Operation Type IT-FN-VEN   
 Contact Andrea Ricci Order Date 1/1/2022 Activity Code   
 Operation Occurred Date 4/2/2023 Requested Delivery Date 4/9/2023 Status Released

Lines Manage Line Order

New Line Delete Line Select items...

Type	No.	Service Tariff No.	Item Reference No.	Incl. in VAT Trans.	Project CM Refers to Period	Description	Location Code	Quantity	Qty. to Assemble to Order	Reserved Quantity	Unit of Measure Code	Unit Price Excl. VAT	Line Discount %	Line Amount Excl. VAT
Item	1996-S			<input type="checkbox"/>		ATLANTA desk		12			PZ	1,404.30		16,851.6
Item	1900-S			<input type="checkbox"/>		PARIS Guest Chair, Black		5			PZ	193.70		968.5
Item	1906-S			<input type="checkbox"/>		ATHENS Desk		1			PZ	435.80		435.8
Charge (Item)	CHARGE			<input type="checkbox"/>		Freight Charge		1				8.00		8.0

Figure 5.7: New feature functionality in sales order

The No. value of the charge item is taken from the extension's setup:

← + ✓ Saved

## Packt Extension Setup

**General**

Minimum Accepted V... 6.00 Default Charge (Item) CHARGE   
 Gift Tolerance Quantit... 2.00

Figure 5.8: Default Charge field in the extension setup

The Default Charge (Item) field is defined as follows:

```
field(4; "Default Charge (Item)"; Code[20])
{
    Caption = 'Default Charge (Item)';
    DataClassification = CustomerContent;
    TableRelation = "Item Charge";
}
```

Here, you saw how you can implement the Handled pattern in an AL business process to make it extendible. In the next section, we'll see how to create a dependent extension (customization for a specific customer) that will use this pattern for extensibility.

## Writing a dependent extension

In the previous chapter and the previous section in this chapter, we developed our base extension and deployed it.

Imagine now that you've deployed this extension to a customer tenant, and the customer now asks you for some customization:

- They want to add the `Certification No` field to the `Vendor Quality` table.
- They want to change the charge calculation for a sales order by using totally custom criteria (for example, a fixed price).

As said before, to create a customization for your customer, you should *never* directly modify your standard extension code; instead, you should create a new extension that will be *dependent* on your base extension.

To do this, we will create a new extension project in Visual Studio Code, which, in this example, we have called `Packt Dependent Extension`. This new extension must be *dependent* on our previously created `Packt Demo Extension` (because we need to extend it).

To declare dependencies between apps, we need to retrieve the `Id`, `name`, `publisher`, and `version` of the base extension (you can check these values from the previously created `app.json` file or directly from the **Extensions Management** page in your Dynamics 365 Business Central tenant, where the main app is installed).

Then, we have to open the `app.json` file of our new extension, go to the `dependencies` block, and insert the details of the dependent extension as follows:

```
"dependencies": [  
  {  
    "id": "dd03d28e-4dfe-48d9-9520-c875595362b6",  
    "name": "Packt Demo Extension",  
    "publisher": "PACKT",  
    "version": "1.0.0.0"  
  }  
],
```

Now, if we download the symbols (AL:Download Symbols), you will see that the symbols of our dependent extension are downloaded into the .alpackages folder in our project:

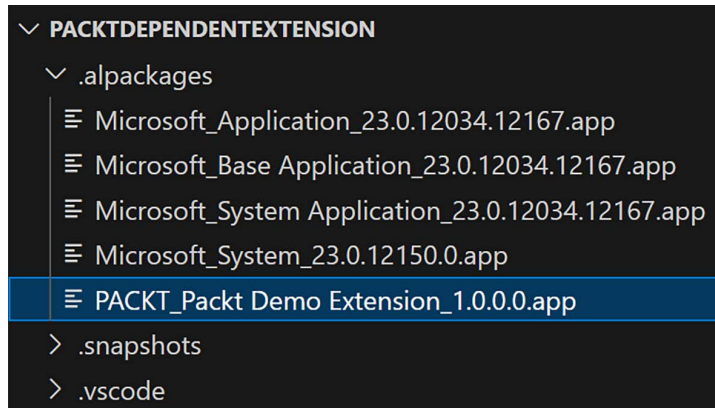


Figure 5.9: Dependent extension downloads in the project folder

The downloaded app file is simply the extension that we've built before. This is the same app file that you use to deploy the app in **Extension Management** or that you deploy using PowerShell, and it is the same file that is generated by Visual Studio Code when you publish your extension.

We're now ready to create our new extension.

To create this new extension (which will be the **per-tenant extension (PTE)** for our customer and contain all the required customizations for that customer), we apply all the rules and best practices used in the main extension. For this PTE, we use the CUST\_ prefix (a sample prefix just to mark all the customer's objects), and we use a dedicated object range.

To add the Certification No. field to Vendor Quality Card, we need to do the following:

- Extend the Vendor Quality table by adding a new field.
- Extend the Vendor Quality Card page to add the new field to the UI.

We can do that because we have the symbols downloaded; otherwise, it's impossible to see the objects defined in another extension.

The tableextension object code that extends the Vendor Quality table is defined as follows:

```
tableextension 50200 CUST_PKTVendorQualityExt extends "PKT Vendor Quality"
{
    fields
    {
        field(50120; "CUST_Certification No."; Text[50])
        {
            Caption = 'Certification No.';
            DataClassification = CustomerContent;
        }
    }
}
```

```

    }
}

```

The `pageextension` object that extends the Vendor Quality Card page is defined as follows:

```

pageextension 50200 CUST_PKTVendorQualityCardExt extends "PKT Vendor Quality Card"
{
    layout
    {
        addlast(General)
        {
            field("CUST_Certification No."; Rec."CUST_Certification No.")
            {
                ApplicationArea = All;
            }
        }
    }
}

```

The second requirement is to customize a standard business process defined in the base extension (Packt Demo Extension). More specifically, the customer wants to have a fixed price for the charge calculation, and this fixed price must be set up (as the default value) in the **Packt Setup** page.

Let's start by adding a Default Charge field to the Packt Setup table and page:

```

tableextension 50201 CUST_PKTPacktSetupExt extends "PKT Packt Setup"
{
    fields
    {
        field(50200; "CUST_Default Charge"; Decimal)
        {
            Caption = 'Default Charge';
            DataClassification = CustomerContent;
        }
    }
}

pageextension 50201 CUST_PKTPacktSetupExt extends "PKT Packt Setup"
{
    layout
    {
        addlast(General)
        {
            field("CUST_Default Charge"; Rec."CUST_Default Charge")
            {

```

```

        ApplicationArea = All;
    }
}
}
}

```

As explained before, we can customize the behavior of a business process defined in another extension only if the base extension has events to subscribe to (you cannot alter the code defined in another extension directly).

To handle extensibility, we used the `Handled` pattern in our base extension. How can we alter the previously created business process (defined in the `PKT Shipment Commission Mgt` codeunit in the `GetShipmentCommission` procedure)?

To skip the standard business process (`GetShipmentCommission`) and add a new custom charge assignment process, we do the following:

1. We subscribe to the `OnBeforeGetShipmentCommission` event.
2. We set the `Handled` parameter to `true`. This ensures that the standard business logic will be skipped because, in the `GetShipmentCommission` procedure, we have used the following code as the first line:

```

OnBeforeGetShipmentCommission(SalesHeader, Handled, total);
    if Handled then
        exit;

```

3. Then, we call our custom business logic from this event subscriber. All of this logic is defined in a codeunit object as follows:

```

codeunit 50200 "CUST_CustomShptCommissionMgt"
{
    [EventSubscriber(ObjectType::Codeunit, Codeunit::"PKT Shipment
Commission Mgt", 'OnBeforeGetShipmentCommission', '', false, false)]
    local procedure CustomChargeCalculationHandler(var SalesHeader:
Record "Sales Header"; var Handled: Boolean; var total: Decimal)
    begin
        total := AssignFixedPrice();
        Handled := true;
    end;

    local procedure AssignFixedPrice(): Decimal
    var
        PacktSetup: Record "PKT Packt Setup";
    begin
        PacktSetup.Get();
        exit(PacktSetup."CUST_Default Charge");
    end;
}

```

When published, we now have two extensions installed (the standard extension and the new customization extension):

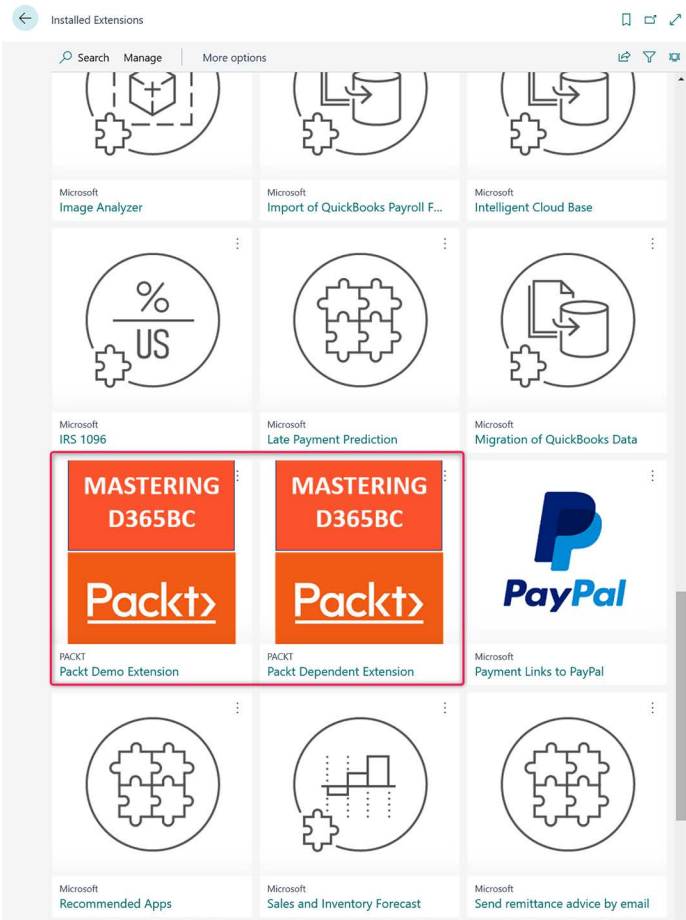


Figure 5.10: Standard and custom extension version

To test whether our customization works, we first need to go to the Packt Extension Setup page to set the fixed charge price (as per business requirements):

← + ✓ Saved

## Packt Extension Setup

---

**General**

Minimum Accepted V...	<input type="text" value="6.00"/>	Default Charge (Item)	<input type="text" value="JB-FREIGHT"/>
Gift Tolerance Quantit...	<input type="text" value="0.00"/>	Default Charge	<input type="text" value="15.00"/>

Figure 5.11: Custom extension setup page

Then, we can go to the same sales order as before and see what happens now when we release it. With only our base app installed, the charge calculation was \$8 (standard algorithm). And now?

If we release the order, a charge (item) line with a 15-dollar value (our fixed price) is created, as expected:

Sales Order

S-ORD101005 · Adatum Corporation

Home Prepare Print/Send Request Approval Order Report Gifts Actions Related Automate Fewer options

Post Release Create Warehouse Shipment Create Inventory Put-away/Pick... Archive Document

**General**

Customer No.	10000	Contact	Robert Townes	Opportunity No.	
Customer Name	Adatum Corporation	No. of Archived Versions	0	Responsibility Center	
Sell-to		Document Date	4/11/2022	Assigned User ID	
Address	192 Market Square	Posting Date	4/11/2022	Status	Released
Address 2		Order Date	4/11/2022	Work Description	
City	Atlanta	Due Date	5/11/2022		
State	GA	Requested Delivery Date			
ZIP Code	31772	Promised Delivery Date			
Country/Region Code	US	External Document No.			
Contact No.	CT000001	Your Reference			
Phone No.		Salesperson Code	JO		
Mobile Phone No.		Campaign No.			
Email	robert.townes@contoso.com				

Lines Manage Line Order Fewer options

Type	No.	Item Reference No.	Description	Location Code	Quantity	Qty. to Assemble to Order	Reserved Quantity	Unit of Measure Code	Unit Price Excl. Tax	Tax Area Code	Tax Group Code	Line Discount %	Line Amount Excl. Tax
Item	1896-S		ATHENS Desk		5			PCS	1,000.80	ATLANTA, GA	FURNITURE		5,004.00
Item	1900-S		PARIS Guest Chair, black		10			PCS	192.80	ATLANTA, GA	FURNITURE		1,928.00
Item	1896-S		ATHENS Desk		1			PCS	1,000.80	ATLANTA, GA	FURNITURE	100	*
Charge (item)	JB-FREIGHT		Freight Charge (JB-Spedition)		1				15.00	ATLANTA, GA	LABOR		15.00

Figure 5.12: Custom charge calculation in the customized extension

We have customized the charge calculation as requested by our customer, without doing code modifications in the base extension.

## Interfaces in AL

In object-oriented programming, an interface is a syntactical contract used to define the capabilities available for a given object, hiding the implementations of these capabilities. Interfaces are a way to achieve polymorphism (the ability to redefine methods for derived classes), and they permit you to write abstract code without relying on the implementation details.

In AL language, interfaces are used to define business processes that can be substituted by other extensions. Interfaces are essentially another method for extensibility. To demonstrate this, we can recreate the same sample we used previously.

Our task is this, then: *Find a way to use interfaces in AL to recreate our previously defined business process to calculate item charges for shipment commissions of sales orders*

When using interfaces in AL, you need to:

- Define an interface object (the contract) that only contains method definitions (no code).
- Define the implementation of this interface (in a codeunit object that *implements* the interface).

As a first step, by using the `tinterface` AL language extension's snippet, we create the following interface object:

```
interface "PKT IShipmentCommissionCalculation"
{
    procedure GetShipmentCommission(SalesHeader: Record "Sales Header"; var
total: Decimal)
}
```

This interface object (called `PKT IShipmentCommissionCalculation`) defines our contract. To handle the shipment commissions calculation process, we need to define an interface method called `GetShipmentCommission` with the two parameters. As you can see, the interface method does not contain any code, only its definition.

Now, we need to define the standard implementation of the `IShipmentCommissionCalculation` interface. The implementation will be defined in a codeunit object, and this must use the `implements` keyword along with the interface name. The implementation is defined as follows:

```
codeunit 50106 "PKT Ship. Commission Calc Base" implements "PKT
IShipmentCommissionCalculation"
{
    //BASE Shipment Commission Calculation
    procedure GetShipmentCommission(SalesHeader: Record "Sales Header"; var
total: Decimal)
    var
        SalesLine: Record "Sales Line";
    begin
        SalesLine.SetRange("Document Type", SalesHeader."Document Type");
        SalesLine.SetRange("Document No.", SalesHeader."No.");
        SalesLine.SetRange(Type, SalesLine.Type::Item);
        if SalesLine.FindSet() then
            repeat
                if SalesLine.Quantity < 10 then
                    total += 1.5
                else
                    total += 5;
            until SalesLine.Next() = 0;
    end;
}
```

This is the standard implementation of the calculation, as explained in the business requirements in the previous section. As you can see, in this implementation, we're not using events. The interface now has business logic, and what is left to do is provide the user with a way to choose the implementation, by linking the interface to an extensible enum.

What we'll do here is the following:

- We will add an option to the extension's setup table that permits users to select the shipment commission calculation process (in other words, selecting the implementation of the standard process or the custom process)
- We'll execute the business process accordingly to the selected implementation of the interface.

To do that, we define the following enum object:

```
enum 50101 "PKT Shipm. Comm. Calc Method" implements "PKT
IShipmentCommissionCalculation"
{
    Extensible = true;

    value(0; Default)
    {
        Implementation = "PKT IShipmentCommissionCalculation" = "PKT Ship.
Commission Calc Base";
    }
}
```

An enum object can implement an interface. If an enum *implements* an interface, every defined value specifies the explicit interface implementer. In this case, when the default value is selected, the implemented interface definition returned is defined in the PKT Ship. Commission Calc Base codeunit.

The enum object is defined as **extensible**. This means that another extension can add a new value to this enum object and so define its own corresponding interface implementation (we'll see this later).

In the extension's setup table (PKT Packt Setup) and on the corresponding card page, we add the following new field:

```
field(10; "Shipmt Commission Calc. Method"; enum "PKT Shipm. Comm. Calc
Method")
{
    Caption = 'Shipment Commission Calc. Method';
    DataClassification = CustomerContent;
}
```

Now, the last step: in our base application (our main extension), we need to call our business process. In the event-based development previously explained, we used the following code:

```
codeunit 50104 "PKT Shipment Commission Mgt"
{

    [EventSubscriber(ObjectType::Codeunit, Codeunit::"Release Sales Document",
'OnBeforeReleaseSalesDoc', '', false, false)]
    local procedure AssignShipmentCommission(var SalesHeader: Record "Sales
Header")
```

```

var
    total: Decimal;
begin
    GetShipmentCommission(SalesHeader, total);
    AddItemCharge(SalesHeader, total);
end;

```

The business process called here is the `GetShipmentCommission` method defined in the `PKT Shipment Commission Mgt` codeunit directly.

When using interfaces, we want to be abstracted from the implementation. We can do that with the following code:

```

[EventSubscriber(ObjectType::Codeunit, Codeunit::"Release Sales Document",
'OnBeforeReleaseSalesDoc', '', false, false)]
    local procedure AssignShipmentCommission(var SalesHeader: Record "Sales
Header")
    var
        PacktSetup: Record "PKT Packt Setup";
        IShipmentCommissionCalculation: Interface "PKT
IShipmentCommissionCalculation";
        total: Decimal;
    begin
        PacktSetup.Get();
        IShipmentCommissionCalculation := PacktSetup."Shipmt Commission Calc.
Method";
        IShipmentCommissionCalculation.GetShipmentCommission(SalesHeader,
total);
        AddItemCharge(SalesHeader, total);
    end;

```

Please note that in the GitHub repo, this code is commented out. If you want to test the interface usage, you need to follow the provided instructions in the source code and uncomment that part.

Here, we read the **Packt Setup** table to retrieve the selected value in the `Shipmt Commission Calc. Method` enum field. This field returns an interface implementation (because it's an *enum* object implementing an interface). Then, we call the `GetShipmentCommission` method of the returned interface object. It can be any implementation returned by the selected enum value; the code is abstracted from the real implementation now.

If we execute this code with only our main extension installed, the returned value from the `Shipmt Commission Calc. Method` enum field is equal to `Default`, and the implementation of the interface to use is defined in the `PKT Ship. Commission Calc Base` codeunit. This means that the `GetShipmentCommission` method defined in this codeunit will be executed.

Note that in the GitHub repo in the PKT Shipment Commission Mgt codeunit, you need to uncomment the interface code and comment the event-base code to test the interfaces.

What happens now if another extension (the previously created dependent extension) wants to handle the shipment commission calculation business process in a totally different way (e.g., a fixed price for orders)?

It needs to simply do the following:

- Add a new interface implementation
- Add a new value to the previously defined enum field with the corresponding interface implementation

To add a new interface implementation, a new codeunit that implements the PKT IShipmentCommissionCalculation interface must be defined:

```
codeunit 50201 "CUST Shipmt Comm. Calc. Custom" implements "PKT
IShipmentCommissionCalculation"
{
    procedure GetShipmentCommission(SalesHeader: Record "Sales Header"; var
total: Decimal)
    begin
        total := AssignFixedPrice()
    end;

    local procedure AssignFixedPrice(): Decimal
    var
        PacktSetup: Record "PKT Packt Setup";
    begin
        PacktSetup.Get();
        exit(PacktSetup."CUST_Default Charge");
    end;
}
```

And we need to add a new value (called **Fixed Price** here) to the Shipmt Commission Calc. Method enum field. This can be done as follows:

```
enumextension 50200 "CUST ShpmtCommissCalcMethodExt" extends "PKT Shpmt. Comm.
Calc Method"
{
    value(50200; "Fixed Price")
    {
        Implementation = "PKT IShipmentCommissionCalculation" = "CUST Shipmt
Comm. Calc. Custom";
    }
}
```

As you can see from the above code, when the enum value is set to `Fixed Price`, the interface implementation that will be returned is defined in the `CUST Shipmt Comm. Calc. Custom codeunit`, so the fixed price process will be called.

By using interfaces, we were able to totally replace a business process simply by adding a new implementation of that process according to the contract.

## Extension's code protection

When you write an extension for Dynamics 365 Business Central, the extension's code is, by default, protected against downloading or debugging.

You can control these settings by using the `resourceExposurePolicy` settings in the extension's manifest file (`app.json`):

```
"resourceExposurePolicy": {  
  "allowDebugging": true,  
  "allowDownloadingSource": true,  
  "includeSourceInSymbolFile": true,  
  "applyToDevExtension": false  
},
```

In the `resourceExposurePolicy` setting, you can specify the following parameters:

- `allowDebugging`: `TRUE` to enable debugging of your extension's code, and `FALSE` otherwise. The default value is `FALSE`. Please remember that if you have marked methods and variables with the `[NonDebuggable]` attribute, these methods and variables will remain non-debuggable.
- `allowDownloadingSource`: `TRUE` to enable the possibility to download the extension's source code and any media file in the extension's package, and `FALSE` otherwise. The default value is `FALSE`.
- `includeSourceInSymbolFile`: `TRUE` if you want the downloaded symbol file in Visual Studio Code, which is accessed by using the **Downloading Symbols** functionality, to contain symbols, source code, and all other resources in the extension package, and `FALSE` otherwise. The default value is `FALSE`.
- `applyToDevExtension`: `TRUE` if you want all resource exposure policy settings specified for the extension to also apply when the extension is released as a dev extension (deployed directly from Visual Studio Code), and `false` otherwise. The default value is `false`.

These settings work at the extension's level.

Sometimes, you may need to change these policies dynamically for specific customer environments (for example, you want to temporarily enable debugging on a specific customer tenant as well if your `resourceExposurePolicy` setting does not permit this).

To override the default `resourceExposurePolicy` setting specified in the extensions's `app.json` file, you need to:

1. Create an instance of the Azure Key Vault service (more information can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/setup-app-key-vault>).
2. Create a secret in the Azure Key Vault instance called `BC-ResourceExposurePolicy-Overrides`.
3. Add a JSON value to this secret by using Azure PowerShell. In this JSON object, you need to specify the tenant IDs for which you want to enable the relative policy.

For example, imagine that I want to temporarily enable the previously mentioned policies for the customer tenant with ID = `6bb8d5f8-91dd-471b-a512-80913c811214`. By using Azure PowerShell, you can do the following:

```
$json = '{
  "allowDebugging": [
    "6bb8d5f8-91dd-471b-a512-80913c811214"
  ],
  "allowDownloadingSource": [
    "6bb8d5f8-91dd-471b-a512-80913c811214"
  ],
  "includeSourceInSymbolFile": [
    "6bb8d5f8-91dd-471b-a512-80913c811214"
  ]
}'
$Secret = ConvertTo-SecureString -String $json -AsPlainText -Force
Set-AzKeyVaultSecret -VaultName "YourKeyVaultName" -Name "BC-
ResourceExposurePolicy-Overrides" -SecretValue $Secret
```

## Summary

In this chapter, we saw how to make our code extensible by using the `Handled` pattern and interfaces, and we saw how to create dependent extensions in order to create customizations of a base application and to create modular solutions.

In the last part of this chapter, we created a new extension that modifies the standard behavior of our base extension, and we looked at the concept of dependency between extensions.

You learned how to create customizations without modifying the base code of your application, and now you're ready to create maintainable and modular applications for Dynamics 365 Business Central.

In the next chapter, we'll see how to handle some advanced topics with AL and the extension model, such as access modifiers and isolated storage, error management, asynchronous programming, JSON and XML management, cryptography, and more.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>





# 6

## Advanced AL Development

In the previous chapter, we saw how to create dependent extensions and how to define interfaces in AL Language to change the behavior of a business process.

In this chapter, we'll focus on other development topics that you need to manage when developing real-world solutions for Dynamics 365 Business Central. These are all separate, advanced features that you can use when creating apps.

This chapter will cover the following topics:

- Understanding immutable keys
- Access modifiers in AL
- Handling errors with TryFunctions
- Using collectible errors and handling actions on errors
- Using Isolated Storage to handle sensitive data
- Handling XML and JSON objects with AL
- Creating and extending Role Center pages
- Creating control add-ins for Dynamics 365 Business Central
- Handling notifications
- Page background tasks and asynchronous programming
- Using Azure Key Vault in AL extensions
- Introducing namespaces in AL Language

By the end of this chapter, you will be able to create better experiences for your users and a more responsive UI, and you will be able to improve the client experience.

### Understanding immutable keys

In Dynamics 365 Business Central, all tables have a (unique) immutable key (a GUID field) that can be used for integration scenarios and to replace the old RecordID property. This field is called `SystemId`, and it's a GUID data type field that specifies a unique, immutable (read-only) identifier for records in a table.

The `SystemId` field (identified with the field number 2000000000 on every table object) has the following characteristics:

- It has a value for every record in a table.
- You can assign a value at insert time; otherwise, the platform automatically assigns one.
- Once `SystemId` has been set, it cannot be changed.
- There is always a unique secondary key in the `SystemId` field.

As a platform rule, modifying `SystemId` of an existing record is not allowed. The `INSERT` function has an override method to handle the `SystemId`:

```
Record.Insert([RunTrigger: Boolean[, InsertWithSystemId: Boolean]])
```

`SystemId` can be manually specified when inserting a new record, as in the following example:

```
myRec.SystemId := '{B6667654-F4B2-B945-8567-006DD6B6775E}';
myRec.Insert(true,true);
```

You can use the `GetBySystemId` function to retrieve a record via its `SystemId`, as in the following example:

```
var
    Customer: Record Customer;
    Text000: Label 'Customer was found.';
begin
    if Customer.GetBySystemId('{B6667654-F4B2-B945-8567-006DD6B6775E}') then
        Message(Text000);
end;
```

You can also set table relations by using the new `SystemId` field, as in the following code:

```
field(1; EntryID; GUID)
{
    DataClassification = CustomerContent;
    TableRelation = Item.SystemId;
}
```

The `SystemID` field is also a key concept on API pages (it can be used as the OData key for records), which we will learn about in *Chapter 13, Dynamics 365 Business Central APIs*.

In the next section, we'll see how to modify the access visibility of AL code.

## Access modifiers in AL

When creating AL extensions for Dynamics 365 Business Central, you can use access modifiers to handle the accessibility level of your AL objects. Access modifiers permit you to control whether an object can be used in your code or code in other modules (dependent extensions).

A table object can have an access level with the following values:

- **Public** (default): The object can be accessed by any other code in the same module and in other modules that reference it.
- **Internal**: The object can be accessed only by code in the same module, but not from another module.

You can change the access level by using the Access property as follows:

```
table 50100 MyTable
{
    DataClassification = CustomerContent;
    Access = Internal;
```

Figure 6.1: Setting the Access property

A field can have an access level with the following values:

- **Public** (default value): The field can be accessed by any other code in the same extension or another extension that references it.
- **Internal**: The field can be accessed only by code in the same extension, but not from another extension.
- **Local**: The field can only be accessed by code in the same table and tableextension *where the field is defined*.
- **Protected**: The field can be accessed only by code in the same table or tableextensions *to that table*.

You can change the Access level of a field in the following way:

```
fields
{
    2 references
    field(1;MyField; Integer)
    {
        DataClassification = CustomerContent;
        Access = Local;
    }
}
```

Figure 6.2: Changing the Access property to Local

If you try to use that field in a different scope than the admitted scope for the specified access level, you will receive an error:

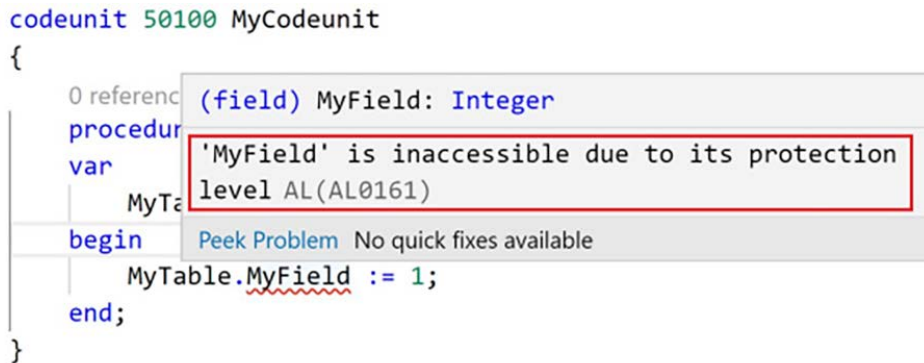


Figure 6.3: Error when using field in the wrong scope

A codeunit object has the following access levels:

- **Public** (default value): The codeunit can be accessed by any other code in the same extension or another extension that references it.
- **Internal**: The codeunit can be accessed only by code in the same extension, but not from another extension.

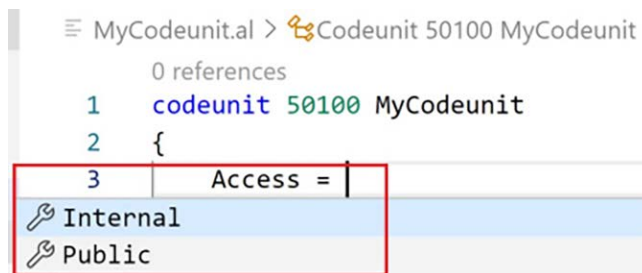


Figure 6.4: Access levels for a codeunit

Also, *procedures* can have an access level (public, local, and internal). You can specify the access level of a procedure in the following way:

```

codeunit 50100 "My Codeunit"
{
    0 references
    internal procedure Foo()
    begin
    end;
}

```

Figure 6.5: Specifying the procedure access level

Please note that *access modifiers are taken into consideration at compile time*.

In AL you can also use **protected variables**. Protected variables are useful to make variables accessible between table and tableextension objects.

As an example, consider the following page object where we declare two variables, where one is a protected variable and the other is a normal variable:

```

page 70102 "My Item List"
{
    Caption = 'My Item List';
    PageType = List;
    SourceTable = Item;

    layout
    {
        area(content)
        {
            repeater(General)
            {
                field("No."; Rec."No.")
                {
                    ApplicationArea = All;
                }
                field(Description; Rec.Description)
                {
                    ApplicationArea = All;
                }
            }
        }
    }

    protected var
        AccessibleVar: Integer; //accessible to page extension

    var
        NotAccessibleVar: Integer; //not accessible to page extension
}

```

Figure 6.6: Page object definition

If we declare a `pageextension` object for this page, only `AccessibleVar` will be visible:

```
pageextension 70122 "My Item List Ext" extends "My Item List"
{
    trigger OnOpenPage()
    begin
        AccessibleVar := 1;
    end;
}
```

Figure 6.7: *pageextension visibility*

## Handling errors with TryFunctions

In traditional programming languages like C#, you can handle errors in a code block by using a try-catch clause:

```
try
{
    //Your code here
}
catch(Exception)
{
    //Error handling here
}
```

In AL language you can use try methods for something similar. A try method permits you to handle errors that occur during a code execution. During a Try method, any database changes (writes) that are executed are not rolled back, allowing the code to continue while also trapping the error and throwing an error.

Let's consider the following piece of AL code written inside a codeunit object:

```
[TryFunction]
local procedure MyTryMethod()
begin
    //Code here
    if (condition) then
        error('An error occurred during the operation');
    end;
```

```
OnRun()  
begin  
    if MyTryMethod then  
        message('Process completed successfully.')  
    else  
        message('Something went wrong on the process.')  
    end;  
end;
```

Here we have defined a procedure called the `MyTryMethod` decorated with the `TryFunction` attribute. In the codeunit's `OnRun` trigger, we execute the process. If the code on `MyTryMethod` fails, the error is trapped, and instead of stopping the execution of the `OnRun` trigger, the execution continues and the *Something went wrong on the process*. message is displayed.

You can also use the `GetLastErrorText` or `GetLastErrorCode` method to obtain the errors that are generated by AL code in your process:

```
String := System.GetLastErrorCode()
```

## Using collectible errors

When creating AL code in a Dynamics 365 Business Central extension, sometimes you need to throw errors to users. The standard way of throwing an error in AL is by using the `ERROR()` construct:

```
ERROR(String [, Value1, ...])
```

When an error occurs, the code stops immediately at the first error it meets and the transaction is rolled back.

Sometimes in a business process, you will have the need to capture multiple error messages and then display them in the UI in a collective way, without stopping the user on every error occurrence. In AL you can do that by using **collectible errors**. By using collectible errors the AL code execution is not stopped on every error occurrence, but you postpone the error handling at the end of your code block.

When using collectible errors, the `ErrorInfo` AL data type defines information about the error. To handle collectible errors, the `ErrorBehavior` attribute of a method specifies the behavior of collectable errors inside the method scope.

Let's consider the following AL code:

```
[ErrorBehavior(ErrorBehavior::Collect)]
1 reference
local procedure RaiseNewError()
var
    myErrorInfo, myErrorInfo2 : ErrorInfo;
    ErrorMessage: Label 'Ahia! There''s an error here!';
    ErrorMessage2: Label 'Ahia! There''s another error here!';
begin
    myErrorInfo := ErrorInfo.Create(ErrorMessage);
    myErrorInfo.ErrorType := ErrorType::Client;
    myErrorInfo.Verboseity := Verboseity::Error;
    myErrorInfo.DetailedMessage := 'This is the detail of the error';
    myErrorInfo.DataClassification := DataClassification::SystemMetadata;
    myErrorInfo.Collectible(true);
    Error(myErrorInfo);
    //...
    myErrorInfo2 := ErrorInfo.Create(ErrorMessage2);
    myErrorInfo2.ErrorType := ErrorType::Client;
    myErrorInfo2.Verboseity := Verboseity::Error;
    myErrorInfo2.DetailedMessage := 'This is the detail of the error';
    myErrorInfo2.DataClassification := DataClassification::SystemMetadata;
    myErrorInfo2.Collectible(true);
    Error(myErrorInfo2);
end;
```

Figure 6.8: Error procedure code

Here I have a procedure called `RaiseNewError` where I'm throwing two different errors and each error has its own details. The procedure uses the `ErrorInfo` data type, which provides a structure for grouping information about an error.

The `Error()` method here has an `ErrorInfo` object as parameter, where the `Collectible` property is set to `true`.

The procedure also has the following attribute defined:

```
ErrorBehavior(Behavior: ErrorBehavior)
```

This specifies the behavior of collectible errors inside the method scope.

What happens now if I execute this code?

This is the result:

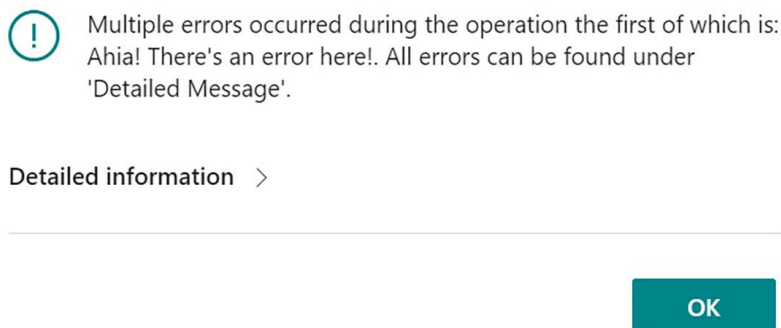


Figure 6.9: Coded error message

An error message appears to the user, showing the first error of the chain together with a message that says that other errors are collected. If you click on the **Detailed information** link, you can find the complete list of errors and the complete call stack:

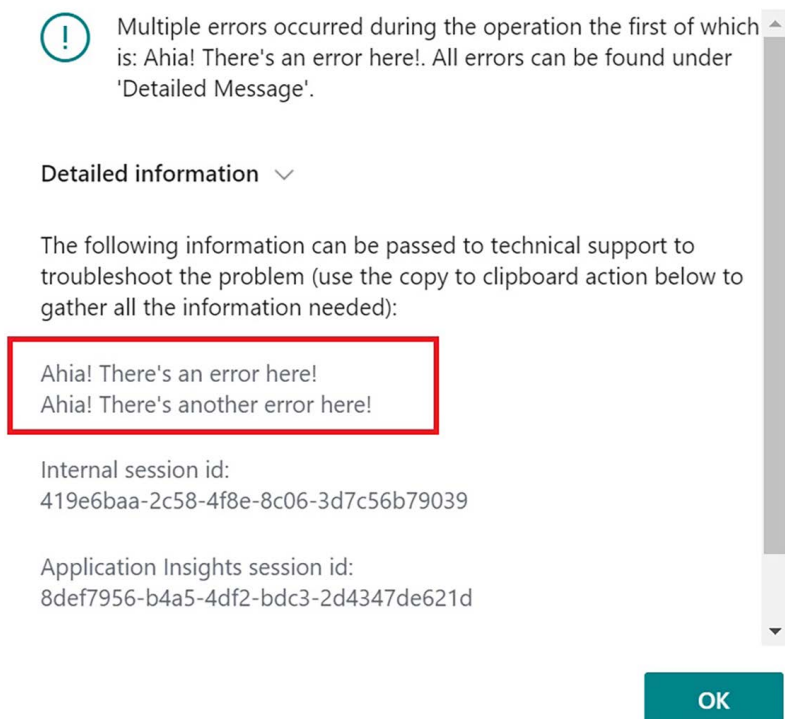


Figure 6.10: List of errors under the Detailed information heading

To handle collectible errors, you also have the following AL methods:

- `System.HasCollectedErrors()`: Gets a value indicating whether errors have been collected in the current error collection scope. This is useful to handle rollbacks.
- `System.GetCollectedErrors([Boolean])`: Gets all collected errors in the current collection scope.
- `System.ClearCollectedErrors()`: Clears all collected errors from the current collection scope.

Please remember that if you clear the list of collected errors (using `ClearCollectedErrors()`), any changes performed in the database won't be rolled back.



If you handle errors in a collectible way, you can check if `System.HasCollectedErrors()` is true and then handle a rollback.

## Handling actions on errors

To improve the user experience on error messages, you can also add custom actions to an error message.

To do that, you can use the `ErrorInfo` object and then use the `AddAction` method:

```
ErrorInfo.AddAction(Caption: Text, CodeunitID: Integer, MethodName: Text)
```

Here, `CodeunitID` is the ID of the codeunit to execute when the action is initiated from the error UI. The codeunit should contain at least one global method to be called by the error action. The global method must have an `ErrorInfo` data type parameter to accept the `ErrorInfo` object.

`MethodName` is the name of the method to execute in the previously defined codeunit.

As an example, consider the following codeunit:

```
codeunit 50000 "PACKT Error Handler"
{
    procedure Handler1(ReceivedErr: ErrorInfo)
    begin
        //...
    end;

    procedure Handler2(ReceivedErr: ErrorInfo)
    begin
        //...
    end;
}
```

You can create a custom error message that displays to users two actions calling the above-defined error handlers in the following way:

```
local procedure RaiseErrorWithActions()
var
    MyErrorInfo: ErrorInfo;
begin
    MyErrorInfo.DataClassification := MyErrorInfo.
DataClassification::SystemMetadata;
    MyErrorInfo.ErrorType := MyErrorInfo.ErrorType::Client;
    MyErrorInfo.Verboseity := MyErrorInfo.Verboseity::Error;
    MyErrorInfo.Title := 'Fatal error!';
    MyErrorInfo.Message := 'An error on the process is happened.';
    MyErrorInfo.AddAction('Start Action 1!', Codeunit::"PACKT Error
Handler", 'Handler1');
    MyErrorInfo.AddAction('Start Action 2!', Codeunit::"PACKT Error
Handler", 'Handler2');
    Error(MyErrorInfo);
end;
```

In the next section we'll see how to extend role centers to satisfy your users' needs.

## Creating and extending role centers

When a user logs in to Dynamics 365 Business Central, they are presented with a page that shows information and actions tailored to their role inside the company. This page is called a **role center**, and it's an integral part of the role-tailored experience of the application.

Dynamics 365 Business Central offers about 20 role centers out of the box (as standard) that you can extend and customize, and you can create new role centers.

A role center is a page that has the PageType property set to RoleCenter. The page structure is as follows:



Figure 6.11: Role center page structure

In the structure diagram, the sections are as follows:

- **Section 1** is the Navigation menu area (one or more items that, when clicked, show other sub-menus). This is used to provide access to the relevant entities for the role to which this Role center page is assigned.
- **Section 2** is the **Navigation Bar** area, which displays a list of links to other pages that will be opened in the content area. This is normally used to add links to the user's most useful entities for their business role, as well as any bookmarked pages.
- **Section 3** is the **Action** area, used to add links to run the most important tasks for this role (links to pages, reports, or codeunits).

- *Section 4* is the **Headline** area, used to display dynamically generated information about the business. We'll see more details about this area in the **Customizing the headline** section of this chapter.
- *Section 5* is the **Wide Cue** area, a set of cues that display numerical values about the business. This area is created with a cuegroup control on a page with `PageType = CardPart`, and with the `Layout` property set to wide.
- *Section 6* is the **Data Cue** area, used to provide a visual representation of aggregated business data (such as KPIs). This part is created with a cuegroup control on a page with `PageType = CardPart`.
- *Section 7* is the **Action Cue** area, which shows tiles that link to some business tasks. This area is created with a cuegroup control on a page with `PageType = CardPart`.
- *Section 8* is the **Chart** area, used to show information as charts (custom business chart control add-ins or embedded Power BI reports).
- *Section 9* is the **CardPart or ListPart page** area, used to display data from the application with a card or a list layout.
- *Section 10* is the **Control add-in** area, used to display custom content using HTML-based control add-ins (written in JavaScript).

A role center page can be created in AL using the following code:

```
page 50101 "My Role Center"
{
    PageType = RoleCenter;
    ApplicationArea = All;

    layout
    {
        area(rolecenter)
        {
            part(SalesPerformance; "Sales Performance")
            {
                Visible = true;
            }

            part(MyCustomers; "My Customers")
            {
                Visible = true;
            }

            part(News;"Headline RC Business Manager")
            {
```

```

        Visible = true;
    }
}
}
}

```

In this example snippet, we have created a role center page with three **parts** (subpages).

You can customize an existing role center page by creating a pageextension object:

```

pageextension 50100 "PKT SalesManagerRoleCenterExt" extends "Sales Manager Role
Center"
{
    layout
    {
        addlast(Content)
        {
            part(MyNews; "PKT MyRoleCenterHeadline")
            {
                ApplicationArea = All;
                Visible = true;
            }
        }
    }

    actions
    {
        addlast(Sections)
        {
            group("PKT My Customers")
            {
                action("Customer Ledger Entries")
                {
                    RunObject = page "Customer Ledger Entries";
                    ApplicationArea = All;
                }
            }
        }
    }
}

```

In this example, just to show the syntax, we have extended the Sales Manager Role Center page by adding a new custom headline part to the content and a new action to open the Customer Ledger Entries page.

Customizing or creating role centers is important because it gives your users a better user experience. In the next section, we'll see how we can customize the Headline part of a role center.

## Customizing the headline

As described earlier, the **headline** is a new part, introduced with the Dynamics 365 Business Central web client, that's used to dynamically display important information about your business. Consider the following screenshot:

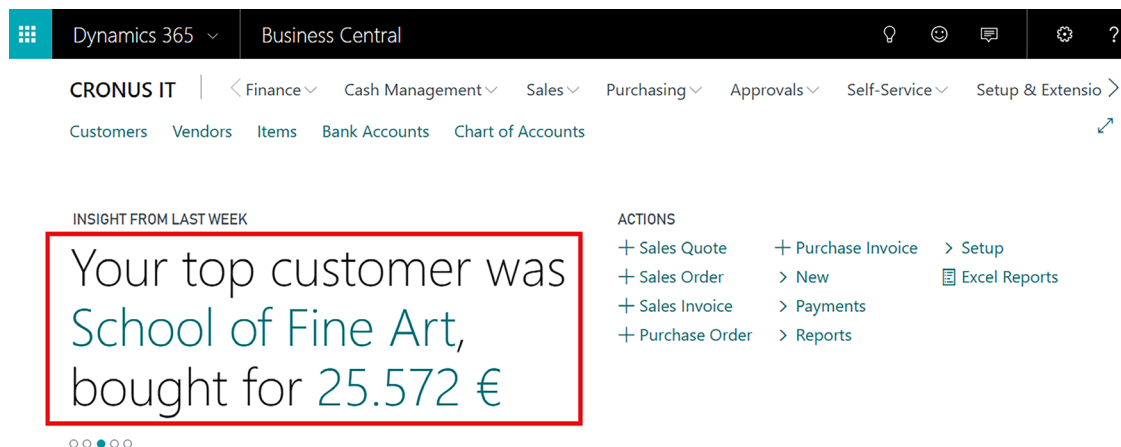


Figure 6.12: Headline page

This is an important part of the Dynamics 365 Business Central role-tailored user experience, and it's recommended to use and customize it to give your users a better experience.

A headline is essentially a page that contains one or more fields (each field is a headline line) and with PageType set as HeadlinePart. This page is only visible inside a role center page.

Dynamics 365 Business Central has nine standard headlines available:

- **Headline RC Business Manager**
- **Headline RC Order Processor**
- **Headline RC Accountant**
- **Headline RC Project Manager**
- **Headline RC Relationship Management**
- **Headline RC Administrator**
- **Headline RC Team Member**
- **Headline RC Production Planner**
- **Headline RC Service Dispatcher**

You can also create your own Headlines by using AL and Visual Studio Code.

A Headline page can be defined in AL as follows:

```

page 50100 "PKT MyRoleCenterHeadline"
{
    PageType = HeadLinePart;
    layout
    {
        area(content)
        {
            field(Headline1; text001)
            {
                ApplicationArea = all;
            }

            field(Headline2; text002)
            {
                ApplicationArea = all;
                trigger OnDrillDown()
                var
                    DrillDownURL: Label 'http://www.demiliani.com';
                begin
                    Hyperlink(DrillDownURL)
                end;
            }

            field(Headline3; text003)
            {
                ApplicationArea = all;
            }

            field(Headline4; text004)
            {
                ApplicationArea = all;
                // Determines visibility while the page is open (custom
criteria)
                Visible=showHeadline4;
            }
        }
    }

    var
        text001: Label 'This is Headline 1';

```

```
text002: Label 'This is Headline 2 (click for details)';
text003: Label 'This is Headline 3';
text004: Label 'This is Headline 4';
showHeadline4: Boolean;

trigger OnOpenPage()
var
    myInt: Integer;
begin
    showHeadline4 := true;
end;
}
```

Here, we have defined a Headline page with four text fields that appear in the Dynamics 365 Business Central UI, with the appropriate text.

In the second headline, we have handled the OnDrillDown event, and if you click on the second headline in this example, you’re redirected to a URL. By handling this event, you can have a clickable headline that shows business details (for example, it can open a Dynamics 365 Business Central detail page). A headline page can also be hidden and visibility can be programmatically set by code (as in Headline 4 in the previous example).

You can edit the formatting of the text displayed on a headline page by using the following expressions:

Expression Tag	Description
<qualifier></qualifier>	This specifies the title that appears above the headline. If it’s not present, the text HEADLINE will be used by default.
<payload></payload>	This specifies the displayed headline text.
<emphasize></emphasize>	The text on this tag is displayed with the biggest size.

Table 6.1 – Expressions for editing headline text

To modify an existing headline, you need to create a pageextension object and extend it. As an example, here we are modifying the standard Headline RC Business Manager page by adding a new headline panel with dynamically created content:

```
pageextension 50101 "PKT MyNewBCHeadline" extends "Headline RC Business
Manager"
{
    layout
    {
        addafter(Control4)
        {
            field(PKTnewHeadlineText;PKTnewHeadlineText)
            {

```

```

        ApplicationArea = all;
    }
}

var
    PKTnewHeadlineText: Text;

trigger OnOpenPage()
var
    HeadlineMgt : Codeunit "Headline Management";
begin
    //Set Headline text
    newHeadlineText := 'This is my new Business Central Headline for '
+ HeadlineMgt.Emphasize('Packt Publishing');
end;
}

```

Here, we've added a new global variable called PKTnewHeadlineText, and this variable is populated in the OnOpenPage trigger of the Headline page with the information that we want to display to our users.

This section explained how to customize the Headline of a role center page and how to show relevant business information to our users. In the next section we'll see how to work with XML and JSON files.

## Handling XML and JSON files with the AL language

The AL language extension has native support to handle XML and JSON documents.

An XML document is represented by using the XmlDocument data type, as explained at <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/xmldocument/xmldocument-data-type>.

The following example code shows how you can import an XML file and load it into an XmlDocument object:

```

local procedure ImportXML()
var
    TempBlob : Codeunit "Temp Blob";
    TargetXmlDoc : XmlDocument;
    XmlDec : XmlDeclaration;
    Instr: InStream;
    filename: Text;
begin
    // Create the XML Document

```

```

TargetXmlDoc := XmlDocument.Create;
xmlDec := xmlDeclaration.Create('1.0', 'UTF-8', '');
TargetXmlDoc.SetDeclaration(xmlDec);

// Create an Instream object & upload the XML file into it
TempBlob.CreateInStream(Instr);
filename := 'data.xml';
UploadIntoStream('Import XML', '', filename, Instr);

// Read stream into new xml document
XmlDocument.ReadFrom(Instr, TargetXmlDoc);
end;

```

Here, we have created an XmlDocument object with an XML declaration, then we have created an InStream object to load the XML file, and we have read the InStream content into the XmlDocument object.

If you reference the TargetXmlDoc object, you will see all of the available methods to handle and manipulate the XML file:

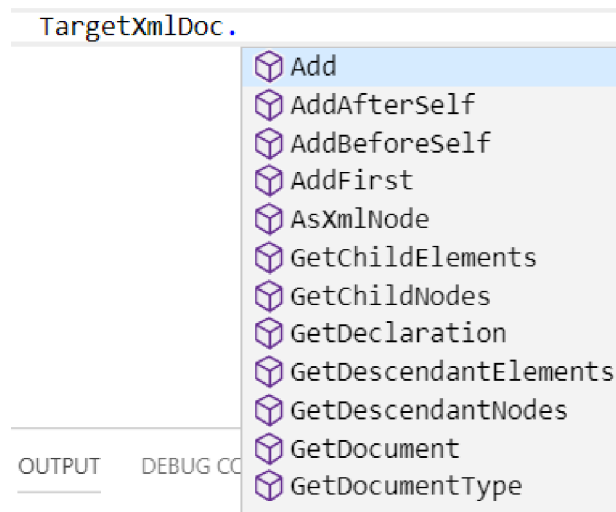


Figure 6.13: XML methods

To create an XML document directly from AL code, you can use the XmlDocument and XmlElement classes:

```

local procedure XMLDocumentCreation()
var
    xmldoc: XmlDocument;
    xmlDec: XmlDeclaration;
    node1: XmlElement;

```

```

        node2: XmlElement;
    begin
        xmlDoc := XmlDocument.Create();
        xmlDec := xmlDeclaration.Create('1.0', 'UTF-8', '');
        xmlDoc.SetDeclaration(xmlDec);
        node1:= XmlElement.Create('node1');
        xmlDoc.Add(node1);
        node2 := XmlElement.Create('node2');
        node2.SetAttribute('ID', '3');
        node1.Add(node2);
    end;

```

This code creates an XML document with a root node (called node1) and a child node (called node2), with an ID attribute that has a value of 3 (<node2 ID="3">).

Native support for JSON documents is provided by using the JsonObject and JsonArray data types. Each of these data types contains the methods to handle a JSON file (both reading and writing) and to manipulate the JSON data (tokens).



A detailed explanation of all of the available methods for these data types can be found at the following links:

- <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/jsonobject/jsonobject-data-type>
- <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/jsonarray/jsonarray-data-type>

The following code shows an example of how to create a JSON representation of a sales order document:

```

procedure CreateJsonOrder(OrderNo: Code[20])
var
    JsonObjectHeader: JsonObject;
    JsonObjectLines: JsonObject;
    JsonOrderArray: JsonArray;
    JsonArrayLines: JsonArray;
    SalesHeader: Record "Sales Header";
    SalesLines: Record "Sales Line";

begin
    //Retrieves the Sales Header
    SalesHeader.Get(SalesHeader."Document Type"::Order, OrderNo);
    //Creates the JSON header details

```

```

JsonObjectHeader.Add('sales_order_no', SalesHeader."No.");
JsonObjectHeader.Add(' bill_to_customer_no', SalesHeader."Bill-to
Customer No.");
JsonObjectHeader.Add('bill_to_name', SalesHeader."Bill-to Name");
JsonObjectHeader.Add('order_date', SalesHeader."Order Date");
JsonOrderArray.Add(JsonObjectHeader);

//Retrieves the Sales Lines
SalesLines.SetRange("Document Type", SalesLines."Document
Type"::Order);
SalesLines.SetRange("Document No.", SalesHeader."No.");
if SalesLines.FindSet then
// JsonObject Init
JsonObjectLines.Add('line_no', '');
JsonObjectLines.Add('item_no', '');
JsonObjectLines.Add('description', '');
JsonObjectLines.Add('location_code', '');
JsonObjectLines.Add('quantity', '');
repeat
    JsonObjectLines.Replace('line_no', SalesLines."Line No.");
    JsonObjectLines.Replace('item_no', SalesLines."No.");
    JsonObjectLines.Replace('description', SalesLines.Description);
    JsonObjectLines.Replace('location_code', SalesLines."Location
Code");
    JsonObjectLines.Replace('quantity', SalesLines.Quantity);
    JsonArrayLines.Add(JsonObjectLines);
until SalesLines.Next() = 0;
JsonOrderArray.Add(JsonArrayLines);
end;

```

This procedure receives an order number as input, retrieves the Sales Header and Sales Line details, and creates a JSON representation. The final result is as follows:

```

[
  {
    "sales_order_no": "S01900027",
    "bill_to_customer_no": "C001435",
    "bill_to_name": "Packt Publishing",
    "order_date": "2019-03-23"
  },
  [
    {

```

```

        "line_no": "10000",
        "item_no": "IT00256",
        "description": "Dynamics 365 Business Central Development Guide",
        "location_code": "MAIN",
        "quantity": 30
    },
    {
        "line_no": "20000",
        "item_no": "IT03465",
        "description": "Mastering Dynamics 365 Business Central",
        "location_code": "MAIN",
        "quantity": 27
    }
]

```

Obviously, you can also receive a JSON representation as input (for example, as a response from an API call) and handle it using the same data types.

## Understanding Isolated Storage

**Isolated Storage** is key-value-based storage that provides data isolation between extensions. Isolated Storage can be used to store data that must be preserved inside the extension scope, and this data is accessible via AL code by using the `IsolatedStorage` data type. The `DataScope` option of the `IsolatedStorage` data type identifies the scope of stored data in Isolated Storage.

`DataScope` is an optional parameter and the default value is `Module`. All possible values are listed in the following table:

Member	Description
Module	It indicates that the record is available in the scope of the app context.
Company	It indicates that the record is available in the scope of the company within the app context.
User	It indicates that the record is available for a user within the app context.
CompanyAndUser	It indicates that the record is available for a user and specific company within the app context.

Table 6.2 – *DataScope* parameter values

To manage data with the `IsolatedStorage` data type, you have the following methods:

Method	Description
<pre>[Ok := ] IsolatedStorage. Set(Key: String, Value: String, [DataScope: DataScope])</pre>	This sets the value associated with the specified key within the extension. The optional <code>DataScope</code> parameter is the scope of the stored data.
<pre>[Ok := ] IsolatedStorage. Get(Key: String, [DataScope: DataScope], var Value: Text)</pre>	This gets the value associated with the specified key within the extension. The optional <code>DataScope</code> parameter is the scope of the data to retrieve.
<pre>HasValue := IsolatedStorage. Contains(Key: String, [DataScope: DataScope])</pre>	This determines whether the storage contains a value with the specified key within the extension. The optional <code>DataScope</code> parameter is the scope to check for the existence of the value with the given key.
<pre>[Ok := ] IsolatedStorage. Delete(Key: String, [DataScope: DataScope])</pre>	This deletes the value with the specified key from the Isolated Storage within the extension. The optional <code>DataScope</code> parameter is the scope to remove the value with the given key.

Table 6.3 – `IsolatedStorage` data type methods

Isolated Storage is useful to store sensitive data, user options, and license keys.

Let’s consider the following example:

```
local procedure IsolatedStorageTest()
var
    keyValue: Text;
begin
    IsolatedStorage.Set('mykey', 'myvalue', DataScope::Company);
```

```

    if IsolatedStorage.Contains('mykey',DataScope::Company) then
    begin
        IsolatedStorage.Get('mykey',DataScope::Company,keyValue);
        Message('Key value retrieved is %1', keyValue);
    end;
    IsolatedStorage.Delete('mykey',DataScope::Company);
end;

```

From the preceding code, we get the following:

1. **IsolatedStorage.Set:** In the first step, we save, in Isolated Storage, a key called mykey with a value of myvalue and DataScope set to Company. The key is visible in the scope of the company within the app context, so no other extensions can access this key.
2. **IsolatedStorage.Get:** In the second step, we check whether a key called mykey is saved in Isolated Storage with DataScope set to Company. If a match is found (key and scope), the key is retrieved (with the Get method) and the value is returned in the keyValue text variable.
3. **IsolatedStorage.Delete:** In the last step, we delete the key for this DataScope.

As previously said, you could also use Isolated Storage to save license keys or license details for your extension. The following code shows how to export the records of a table called License to JSON, then how to encrypt the JSON value, and finally, how to store the encrypted text in Isolated Storage:

```

local procedure StoreLicense()
var
    StorageKey: Text;
    LicenseText: Text;
    EncryptManagement: Codeunit "Cryptography Management";
    License: Record License temporary;

begin
    StorageKey := GetStorageKey();
    LicenseText := License.WriteLicenseToJson();
    if EncryptManagement.IsEncryptionEnabled() and EncryptManagement.
IsEncryptionPossible() then
        LicenseText := EncryptManagement.Encrypt(LicenseText);
    if IsolatedStorage.Contains(StorageKey, DataScope::Module) then
        IsolatedStorage.Delete(StorageKey);
    IsolatedStorage.Set(StorageKey, LicenseText, DataScope::Module);
end;

local procedure GetStorageKey(): Text
var

```

```

        //Returns a GUID
        StorageKeyTxt: Label 'dd03d28e-4acb-48d9-9520-c854495362b6', Locked =
true;
    begin
        exit(StorageKeyTxt);
    end;

    local procedure ReadLicense()
    var
        StorageKey: Text;
        LicenseText: Text;
        EncryptManagement: Codeunit "Cryptography Management";
        License: Record License temporary;
    begin
        StorageKey := GetStorageKey();
        if IsolatedStorage.Contains(StorageKey, DataScope::Module) then
            IsolatedStorage.Get(StorageKey, DataScope::Module, LicenseText);
        if EncryptManagement.IsEncryptionEnabled() and EncryptManagement.
IsEncryptionPossible() then
            LicenseText := EncryptManagement.Decrypt(LicenseText);
        License.ReadLicenseFromJson(LicenseText);
    end;

```

Here, the License table is declared as a temporary table. This way, the data is isolated in the calling codeunit.

Related to secret management with Isolated Storage, remember that:

- In Dynamics 365 Business Central SaaS, sensitive data stored in Isolated Storage is always encrypted.
- In Dynamics 365 Business Central on-premises, encryption is controlled by the end user (via the **Data Encryption Management** page):
  - If encryption is turned on, a secret stored in Isolated Storage is automatically encrypted.
  - A secret that was inserted while encryption was turned off will remain unencrypted if encryption is turned on.
  - If you turn off encryption, the secret will be decrypted.

According to these changes, if you have an extension that works for Dynamics 365 Business Central SaaS and on-premises and you're using Isolated Storage to store secrets, you need to check whether encryption is enabled (which is always true for SaaS) and then save the secret accordingly.

So, a function that saves a license key to Isolated Storage and works for Dynamics 365 Business Central SaaS and on-premises will be as follows:

```
local procedure StoreLicense()
var
    licenseKeyValue: Text;
begin
    if not EncryptionEnabled() then
        IsolatedStorage.Set('LicenseKey',licenseKeyValue,DataScope::Module)
    else
        IsolatedStorage.
SetEncrypted('LicenseKey',licenseKeyValue,DataScope::Module)
end;
```

With the SetEncrypted method, you can now automatically save a secret by using encryption (no more calls to the Cryptography Management codeunit).

We have seen how to use Isolated Storage to improve data security in our extensions. In the next section, we'll see how to create control add-ins.

## Working with control add-ins

**Control add-in** objects are a way to add custom functionalities (functions or UI customizations) to the Dynamics 365 Business Central client. A control add-in can interact with Dynamics 365 Business Central events and raise events for your AL code.

A control add-in can be defined in AL code by using the `tcontroladdin` snippet, which has the following structure:

```
controladdin MyControlAddIn
{
    RequestedHeight = 300;
    MinimumHeight = 300;
    MaximumHeight = 300;
    RequestedWidth = 700;
    MinimumWidth = 700;
    MaximumWidth = 700;
    VerticalStretch = true;
    VerticalShrink = true;
    HorizontalStretch = true;
    HorizontalShrink = true;
    Scripts =
        'script1.js',
        'script2.js';
```

```

StyleSheets =
    'style.css';
StartupScript = 'startupScript.js';
RecreateScript = 'recreateScript.js';
RefreshScript = 'refreshScript.js';
Images =
    'image1.png',
    'image2.png';

event MyEvent()

procedure MyProcedure()
}

```

As you can see from the code snippet, when you define a `controladdin` object, you need to set the `Scripts` property to include the scripts of your control add-in (in a JavaScript file). These scripts can be local `.js` files or external files referenced via HTTP or HTTPS.

The `StartupScript` property permits you to call a script that must be executed when the page that hosts the `controladdin` object is loaded. You can style your `controladdin` object by using the `StyleSheet` property (which permits you to reference a CSS file) and the `Images` property (which permits you to load images into your add-in).



When defining the style of a `controladdin` object in Dynamics 365 Business Central, please always refer to the *Control Add-in Style Guide* at the following link: <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-control-addin-style>.

## Creating a PDF-Viewer control add-in

In this section we'll see how we can integrate a control add-in into a Dynamics 365 Business Central page. The goal is to create a PDF reader and use it to display PDF attachments to Dynamics 365 Business Central documents directly in the browser (without downloading the file).

To accomplish this task, we'll use the free `PDF.js` JavaScript library available here: <https://mozilla.github.io/pdf.js/>

You can find the source code of this example in the book's GitHub repository here: <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition.git>

As a first step, we declare in AL a `controladdin` object and add some JavaScript and CSS files to our extension. In the `script.js` file we place all the JavaScript-based logic to handle the add-in rendering and visibility functionalities (describing the JavaScript code is out of scope here).

The controladdin object is defined as follows:

```
controladdin "PACKT PDF Viewer"
{
    Scripts = 'https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.
js', 'https://cdnjs.cloudflare.com/ajax/libs/pdf.js/3.3.122/pdf.min.js',
'JavaScript/script.js';
    StartupScript = 'JavaScript/Startup.js';
    StyleSheets = 'JavaScript/stylesheet.css';

    MinimumHeight = 1;
    MinimumWidth = 1;
    MaximumHeight = 2000;
    HorizontalStretch = true;
    VerticalStretch = true;
    VerticalShrink = true;
    HorizontalShrink = true;
    event ControlAddinReady();
    event onView()
    procedure LoadPDF(PDFDocument: Text; IsFactbox: Boolean)
    procedure SetVisible(IsVisible: Boolean)
}
```

In our extension we have created a page (called *PACKT PDF Viewer Doc. Attachmt*) that adds the control add-in to a page field and displays the content of a PDF file in the browser by using the add-in. This can be done by using the usercontrol object as follows:

```
page 60000 "PACKT PDF Viewer Doc. Attachmt"
{
    Caption = 'PDF Viewer';
    PageType = Card;
    UsageCategory = None;
    SourceTable = "Document Attachment";
    layout
    {
        area(content)
        {
            group(General)
            {
                ShowCaption = false;
            }
        }
    }
}
```

```

        usercontrol(PDFViewer; "PACKT PDF Viewer")
    {
        ApplicationArea = All;

        trigger ControlAddinReady()
        begin
            SetPDFDocument();
        end;
    }
}
}
}
}

```

In the ControlAddinReady trigger we call a procedure called SetPDFDocument, where we load a PDF file into the PDF viewer control. The PDF (saved as a document attachment in Dynamics 365 Business Central) is available via the Document Reference ID field (a field of type Media) of the Document Attachment table. The procedure is defined as follows:

```

local procedure SetPDFDocument()
var
    Base64Convert: Codeunit "Base64 Convert";
    TempBlob: Codeunit "Temp Blob";
    InStreamVar: InStream;
    OutStreamVar: OutStream;
    PDFAsTxt: Text;
begin
    CurrPage.PDFViewer.SetVisible(Rec."Document Reference ID".HasValue());
    if not Rec."Document Reference ID".HasValue() then
        exit;

    TempBlob.CreateInStream(InStreamVar);
    TempBlob.CreateOutStream(OutStreamVar);
    Rec."Document Reference ID".ExportStream(OutStreamVar);

    PDFAsTxt := Base64Convert.ToBase64(InStreamVar);

    CurrPage.PDFViewer.LoadPDF(PDFAsTxt, false);
end;

```

Then we extend the Document Attachment Details page to add a View PDF action. The new action is defined as follows:

```
pageextension 60000 "PACKT DocumentAttachmentDetExt" extends "Document
Attachment Details"
{
    actions
    {
        addlast(processing)
        {
            action("PACKT View PDF")
            {
                ApplicationArea = All;
                Image = Text;
                Caption = 'View PDF';
                ToolTip = 'View PDF';
                Promoted = true;
                PromotedOnly = true;
                PromotedCategory = Process;
                Enabled = Rec."File Extension" = 'pdf';
                trigger OnAction()
                var
                    PDFViewerDocAttachment: Page "PACKT PDF Viewer Doc.
Attachmnt";
                begin
                    PDFViewerDocAttachment.SetRecord(Rec);
                    PDFViewerDocAttachment.SetTableView(Rec);
                    PDFViewerDocAttachment.Run();
                end;
            }
        }
    }
}
```

What is the result of this implementation?

We can open Dynamics 365 Business Central, select a *Sales Order* document, and add a PDF document via the standard attachment feature. As you can see from the below image, now we have a new *View PDF* action:

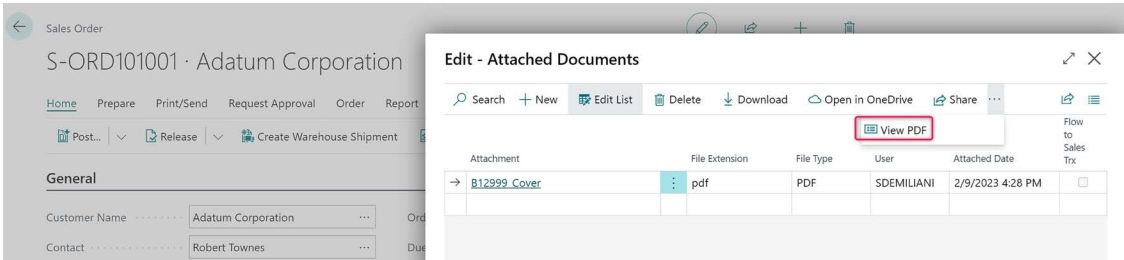


Figure 6.14: Implemented View PDF action

If you click the View PDF action, the attached *Media* file is retrieved from the selected record, its stream is passed to the control add-in, and the PDF file is displayed in the browser:

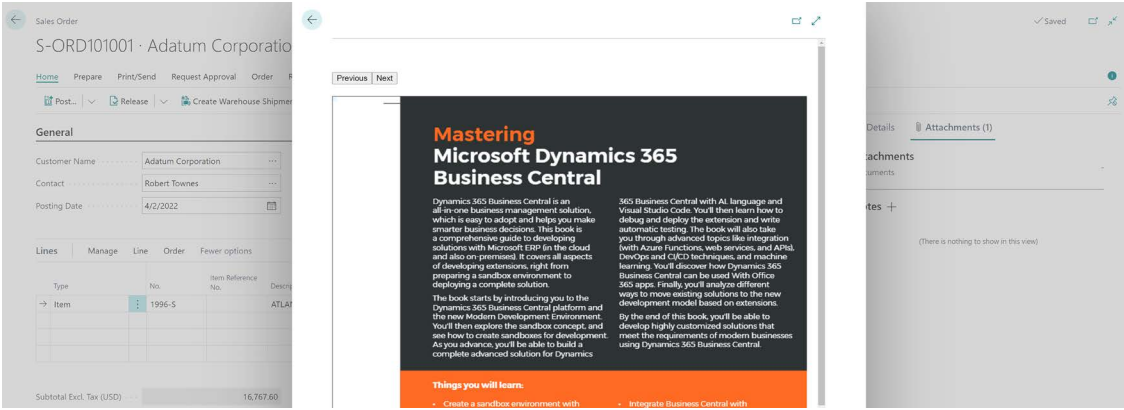


Figure 6.15: PDF file displayed in a browser

# Notifications inside Dynamics 365 Business Central

Dynamics 365 Business Central permits you to programmatically send non-intrusive notifications to your users inside the web client UI to display information, messages, or error notifications. These notifications are non-modal, so they don't require your users to stop working and perform some action on the notification message immediately. They can also be dismissed if necessary.

Notifications appear in the **notification bar** at the top of the page in which the user is currently working. The application can send multiple notifications to the user, and they will all appear in the notification bar in chronological order. They will remain in the notification bar until the user takes action on them or dismisses them or for the duration of the page instance.

As a developer, you can programmatically create notifications by using the Notification and NotificationScope AL objects.

As an example of how to use Notifications inside Dynamics 365 Business Central, we will create a notification on the Purchase Order page that appears if the selected Vendor has a balance due.

The pageextension object is defined as follows:

```
pageextension 50100 PurchaseOrderExt extends "Purchase Order"
{
    trigger OnOpenPage()
    var
        Vendor: Record Vendor;
        VendorNotification: Notification;
        OpenVendor: Text;
        TextNotification: Label 'This Vendor has a Balance due. Please check
before sending orders.';
        TextNotificationAction: Label 'Check balance due';
    begin
        Vendor.Get("Buy-from Vendor No.");
        Vendor.CalcFields("Balance Due");
        if Vendor."Balance Due" > 0 then begin
            VendorNotification.Message(TextNotification);
            VendorNotification.Scope := NotificationScope::LocalScope;
            VendorNotification.SetData('VendorNo', Vendor."No.");
            VendorNotification.AddAction(TextNotificationAction,
Codeunit::ActionHandler, 'OpenVendor');
            VendorNotification.Send();
        end;
    end;
}
```

We check whether the vendor has a balance due. If so, a Notification object is created. A Notification object has a Message property (which defines the content of the notification that will appear in the UI) and Scope. Now, Scope defines where the message will appear to the user, and it could be one of the following:

- LocalScope (default): The notification will appear on the user's current page and go away as soon as the page closes.
- GlobalScope (not currently supported): The notification will appear regardless of which page the user is working on.

When defining the Notification object, we use the SetData method to set a data property value to the notification (in this case, the Vendor number), and we use the AddAction method to add an action to the notification message (we want an action that immediately opens the Vendor Card page). The AddAction method starts a method called OpenVendor, defined in a codeunit called ActionHandler.

The codeunit object for this is defined as follows:

```
codeunit 50100 ActionHandler
{
```

```
procedure OpenVendor(VendorNotification: Notification)
var
    VendorCode: Text;
    Vendor: Record Vendor;
    VendorCard: Page "Vendor Card";
begin
    VendorCode := VendorNotification.GetData('VendorNo');
    if Vendor.Get(VendorCode) then begin
        VendorCard.SetRecord(Vendor);
        VendorCard.Run();
    end;
end;
}
```

Here, when the action inside the notification is clicked, the code retrieves the VendorNo parameter from the Notification object, retrieves the Vendor record, and opens the Vendor Card by passing the retrieved record.

When you open a purchase order from Dynamics 365 Business Central and the selected vendor has a balance due, you will now see the following notification:

← PURCHASE ORDER

+

106001 · Fabrikam, Inc.

Notifications: 2

✕ This Vendor has a Balance due. Please check before sending orders. [Check balance due](#)

✕ Reminder: your work date is 4/8/2019 [Use today](#) | [Change to...](#) | [Turn off reminder](#)

General

Show more

Vendor Name

Fabrikam, Inc.

Vendor Invoice No.

5755

Contact

Krystal York

Vendor Shipment...

Document Date

4/8/2019

Lines

Manage

More options

Type	No.	Description	Location Code	Bin Code
→ Item	1896-S	ATHENS Desk		

Figure 6.16: Implemented notification

If you click on the *Check balance due* action inside the notification, the Vendor Card page with the selected vendor record is opened and the user can act accordingly.

Notifications are extremely important to use when you create extensions for Dynamics 365 Business Central because they will permit you to give a better experience to your users.

In the next section, we'll see how to use asynchronous programming inside Dynamics 365 Business Central.

## Understanding page background tasks

Dynamics 365 Business Central has a new feature to handle asynchronous programming called **page background tasks**.

**Page background tasks** permit you to define a read-only and long-running process on a page that can be executed asynchronously in a background thread (isolated from the parent session). You can start the task and continue working on the page without waiting for the task to complete. The following diagram (provided by Microsoft on the official documentation page at the following link: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-page-background-tasks>) shows the flow of a background task:

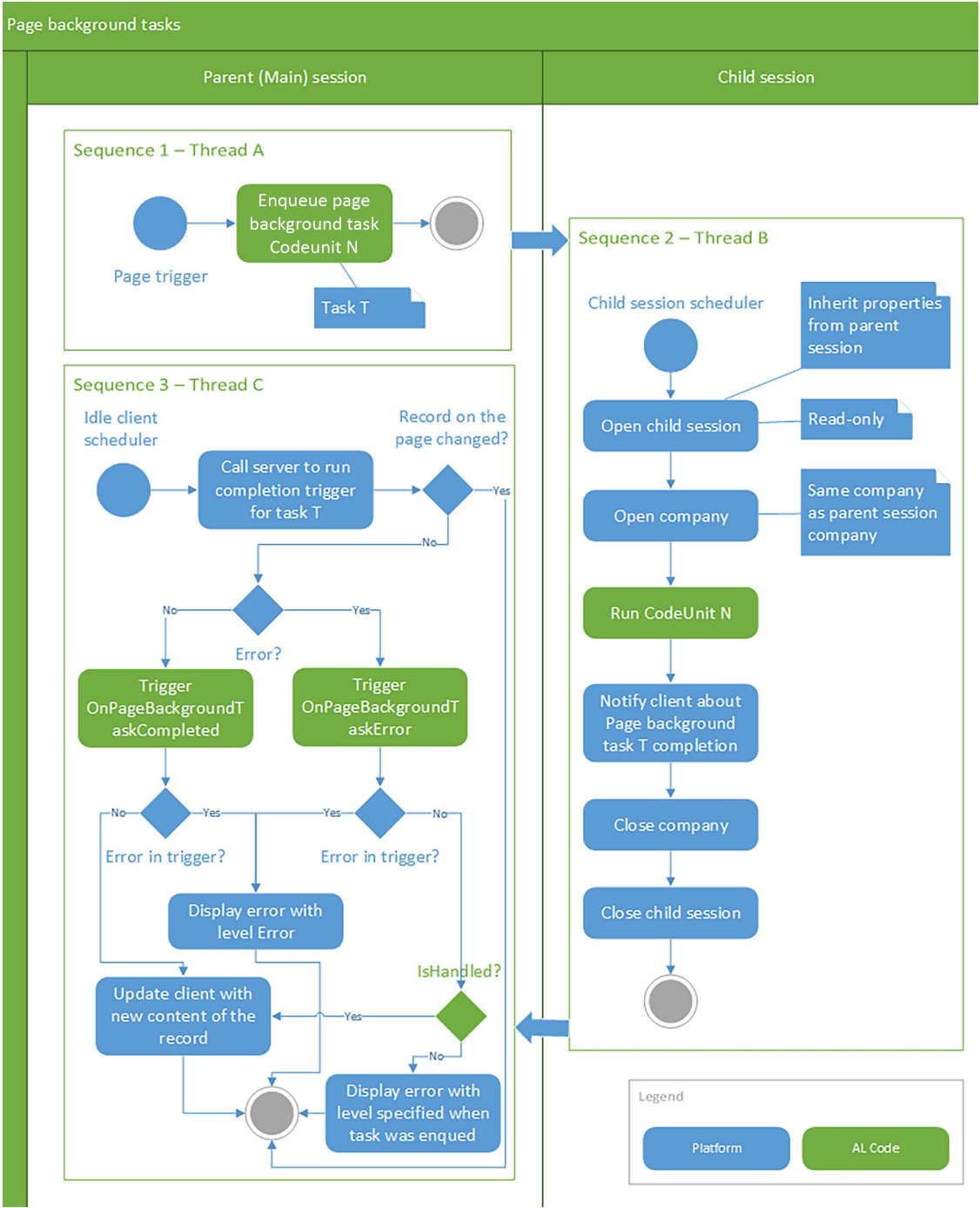


Figure 6.17: Background task flow diagram

A page background task has the following properties:

- It's a read-only session (it cannot write or lock the database).
- It can be canceled and has a default and maximum timeout.
- Its lifetime is controlled by the current record (it is canceled when the current record is changed, the page is closed, or the session ends).
- Completion triggers are invoked on the page session (such as updating page records and refreshing the UI).
- It can be queued up.
- The parameters passed to and returned from page background tasks are in the form of a `Dictionary<string, string>` object.
- The callback triggers cannot execute operations on the UI, except notifications and control updates.
- There is a limit on the number of background tasks per session (if the limit is reached, the tasks are queued).

To create a page background task, the basic steps are as follows:

1. Create a codeunit that contains the business logic to execute in the background.
2. On the page where the task must be started, do the following:
  - a. Add code that creates the background task (`EnqueueBackgroundTask`).
  - b. Handle the task completion results by using the `OnPageBackgroundTaskCompleted` trigger (this is where you can update the page UI).
  - c. You can also use the `OnPageBackgroundTaskError` trigger to handle possible task errors.

Here is an example of how to implement the preceding logic. In the **Customer Card**, we want to execute a background task that calculates some sales statistics values for the selected customer without blocking the UI (in a more complex scenario, imagine retrieving this data from an external service).

Our calling page (`Customer Card`) passes a `Dictionary<string, string>` object (a key-value pair) to the background task, with the key set to `CustomerNo` and the value set to the `No.` field of our selected **Customer** record. The task codeunit retrieves the `CustomerNo` value, calculates the total sales amount for this customer, the number of items sold, and the number of items shipped, and returns a `Dictionary<string><string>` object with the key set to `TotalSales` and a value that is the calculated sales amount.

The task codeunit is defined as follows:

```
codeunit 50105 TaskCodeunit
{
    trigger OnRun()
    var
        Result: Dictionary of [Text, Text];
        CustomerNo: Code[20];
```

```

    CustomerSalesValue: Text;
    NoOfSalesValue: Text;
    NoOfItemsShippedValue: Text;
begin
    CustomerNo := Page.GetBackgroundParameters().Get('CustomerNo');
    if CustomerNo = '' then
        Error('Invalid parameter CustomerNo');
    if CustomerNo <> '' then begin
        CustomerSalesValue := Format(GetCustomerSalesAmount(CustomerNo));
        NoOfSalesValue := Format(GetNoOfItemsSales(CustomerNo));
        NoOfItemsShippedValue := Format(GetNoOfItemsShipped(CustomerNo));
        //sleep for demo purposes
        Sleep((Random(5)) * 1000);
    end;
    Result.Add('TotalSales', CustomerSalesValue);
    Result.Add('NoOfSales', NoOfSalesValue);
    Result.Add('NoOfItemsShipped', NoOfItemsShippedValue);
    Page.SetBackgroundTaskResult(Result);
end;

local procedure GetCustomerSalesAmount(CustomerNo: Code[20]): Decimal
var
    SalesLine: Record "Sales Line";
    amount: Decimal;
begin
    SalesLine.SetRange("Document Type", SalesLine."Document Type"::Order);
    SalesLine.SetRange("Sell-to Customer No.", CustomerNo);
    if SalesLine.FindSet() then
        repeat
            amount += SalesLine."Line Amount";
        until SalesLine.Next() = 0;
    exit(amount);
end;

local procedure GetNoOfItemsSales(CustomerNo: Code[20]): Decimal
var
    SalesLine: Record "Sales Line";
    total: Decimal;
begin
    SalesLine.SetRange("Document Type", SalesLine."Document Type"::Order);
    SalesLine.SetRange("Sell-to Customer No.", CustomerNo);

```

```

SalesLine.SetRange(Type, SalesLine.Type::Item);
if SalesLine.FindSet() then
    repeat
        total += SalesLine.Quantity;
    until SalesLine.Next() = 0;
    exit(total);
end;

local procedure GetNoOfItemsShipped(CustomerNo: Code[20]): Decimal
var
    SalesShipmentLine: Record "Sales Shipment Line";
    total: Decimal;
begin
    SalesShipmentLine.SetRange("Sell-to Customer No.", CustomerNo);
    SalesShipmentLine.SetRange(Type, SalesShipmentLine.Type::Item);
    if SalesShipmentLine.FindSet() then
        repeat
            total += SalesShipmentLine.Quantity
        until SalesShipmentLine.Next() = 0;
        exit(total);
    end;
}

```

Then, we create a pageextension object to extend the Customer Card to add the new SalesAmount, NoOfSales, and NoOfItemsShipped fields (calculated by the background task) and to add code to start the task and read the results. The pageextension object is defined as follows:

```

pageextension 50105 CustomerCardExt extends "Customer Card"
{
    layout
    {
        {
            addlast(General)
            {
                field(SalesAmount; SalesAmount)
                {
                    ApplicationArea = All;
                    Caption = 'Sales Amount';
                    Editable = false;
                }
                field(NoOfSales; NoOfSales)
                {
                    ApplicationArea = All;

```

```

        Caption = 'No. of Sales';
        Editable = false;
    }
    field(NoOfItemsShipped; NoOfItemsShipped)
    {
        ApplicationArea = All;
        Caption = 'Total of Items Shipped';
        Editable = false;
    }
}
}
var
    // Global variable used for the TaskID
    TaskSalesId: Integer;
    // Variables for the sales amount field (calculated from the background
task)
    SalesAmount: Decimal;
    NoOfSales: Decimal;
    NoOfItemsShipped: Decimal;

trigger OnAfterGetCurrRecord()
var
    TaskParameters: Dictionary of [Text, Text];
begin
    TaskParameters.Add('CustomerNo', Rec."No.");
    CurrPage.EnqueueBackgroundTask(TaskSalesId, Codeunit::TaskCodeunit,
TaskParameters, 20000, PageBackgroundTaskErrorLevel::Warning);
end;

trigger OnPageBackgroundTaskCompleted(TaskId: Integer; Results: Dictionary
of [Text, Text])
var
    PBTNotification: Notification;
begin
    if (TaskId = TaskSalesId) then begin
        Evaluate(SalesAmount, Results.Get('TotalSales'));
        Evaluate(NoOfSales, Results.Get('NoOfSales'));
        Evaluate(NoOfItemsShipped, Results.Get('NoOfItemsShipped'));
        PBTNotification.Message('Sales Statistics updated.');
```

```

trigger OnPageBackgroundTaskError(TaskId: Integer; ErrorCodes: Text;
ErrorText: Text; ErrorCallStack: Text; var IsHandled: Boolean)
var
    PBTErrNotification: Notification;
begin
    if (ErrorText = 'Invalid parameter CustomerNo') then begin
        IsHandled := true;
        PBTErrNotification.Message('Something went wrong. Invalid
parameter CustomerNo. ');
        PBTErrNotification.Send();
    end
    else
        if (ErrorText = 'Child Session task was terminated because of a
timeout.') then begin
            IsHandled := true;
            PBTErrNotification.Message('It took to long to get results.
Try again. ');
            PBTErrNotification.Send();
        end
    end;
}

```

In the `OnAfterGetCurrRecord` trigger, we add the parameters required to start our background task and call the `EnqueueBackgroundTask` method. The `EnqueueBackgroundTask` method creates and queues a background task that runs the specified codeunit (without a UI) in a read-only child session of the page session. If the task completes successfully, the `OnPageBackgroundTaskCompleted` trigger is invoked. If an error occurs, the `OnPageBackgroundTaskError` trigger is invoked. If the page is closed before the task completes, or the page record ID on the task changes, the task is canceled.

In the `OnPageBackgroundTaskCompleted` trigger, we retrieve the `TotalSales` parameter from the dictionary, and the UI (the relative field on the page) is updated accordingly.

We have seen how to use the new asynchronous programming features inside Dynamics 365 Business Central pages. This is an important feature that improves general application performance in many scenarios.

## Using Azure Key Vault in AL extensions

Azure Key Vault is a cloud service offered by the Azure platform that permits you to securely store secrets (like keys, certificates, etc.) for your applications in a centralized cloud store.

When developing AL extensions, you may need to handle secrets (for example, a password to access an external system), and the Azure Key Vault service is a recommended secure store to use in these scenarios.

An AL extension can retrieve secrets from one or two instances of Azure Key Vault.

To use Azure Key Vault from an AL extension, you need to first create an instance of the Azure Key Vault service in your Azure subscription.

To do that:

1. From the Azure portal menu, select **Create a resource**.
2. In the search box, enter **Key Vault**.
3. From the results list, choose **Key Vault**.
4. In the **Key Vault** section, choose **Create**.
5. In the **Create key vault** section, provide the following information:
  - **Name:** Your key vault name. A unique name is required.
  - **Subscription:** Select a subscription.
  - Under **Resource Group**, choose **Create new** and enter a resource group name.
  - In the **Location** pull-down menu, choose a location.
  - Leave the other options to their defaults.
6. Select **Create**.

When the key vault instance is created, take note of these two properties:

- **Vault Name:** The name of your newly created key vault.
- **Vault URI:** The URI of your key vault.

When the Azure Key Vault instance is created, you need to set up Dynamics 365 Business Central to connect to that key vault instance. This setup process is different between Dynamics 365 Business Central SaaS and on-premises versions. The steps to complete this setup process are described in the following Microsoft documentation:

- **Setting up App Key Vaults for Business Central Online:** <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/setup-app-key-vault>
- **Setting up App Key Vaults for Business Central On-premises:** <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/setup-app-key-vault-onprem>

Please remember that for Dynamics 365 Business Central online, the app key vault feature is only supported for AppSource extensions.

When all is ready, you can specify the key vaults for your extension in the `app.json` file by using the `keyVaultUrls` property as follows:

```
"keyVaultUrls": [ "https://mykeyvault.vault.azure.net" ]
```

By using this property you can specify up to two key vault instances. You should use two key vault instances created in two different Azure regions if you want secrets to be highly available. At runtime, the Business Central platform will iterate both key vaults until the secret is successfully retrieved.

To read key vault secrets at runtime from AL code you can use the App Key Vault Secret Provider codeunit in the Secrets module of the **System Application**. An example is the following:

```
[NonDebuggable]
internal procedure ReadSecrets()
var
    SecretProvider: Codeunit "App Key Vault Secret Provider";
    SecretValue: Text;
begin
    if SecretProvider.TryInitializeFromCurrentApp() then begin
        if SecretProvider.GetSecret('MySecret', SecretValue) then
            Message('Retrieved secret: ' + SecretValue)
        else
            Message('Failed to retrieve secret')
        end
    else
        Message('ERROR: ' + GetLastErrorText());
    end;
```

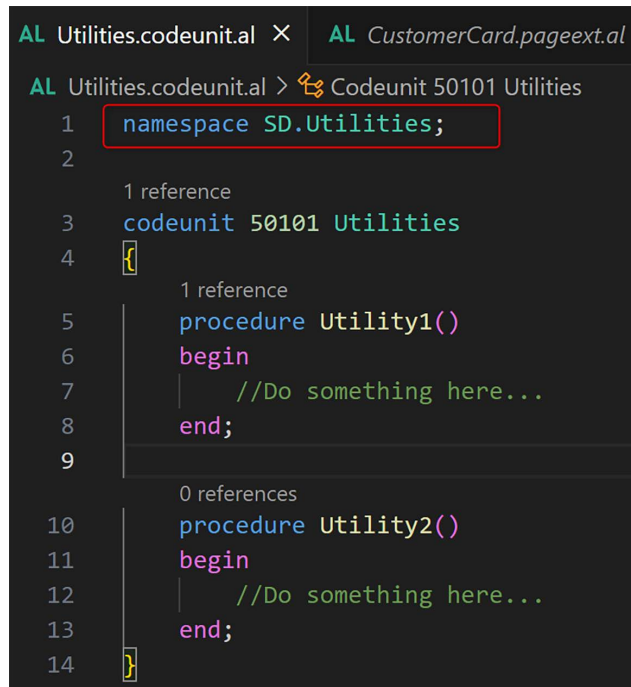
Here we call the TryInitializeFromCurrentApp method to initialize the codeunit with the key vaults specified in the extension's manifest (app.json) file. Then we use the GetSecret method to retrieve the value of the specified secret.

To improve the security of a method that reads secrets, we mark the procedure with the [NonDebuggable] attribute to prevent other partners from debugging the code and then reading the secrets.

## Namespaces in AL language

Starting from the Dynamics 365 Business Central 2023 Wave 2 release (version 23), the AL language supports namespaces. Namespaces permit you to organize objects and avoid naming conflicts between extensions.

When creating a new extension, now you can declare that an object belongs to a namespace like in the following example:

The image is a screenshot of the Microsoft Dynamics 365 Business Central AL language editor. At the top, there are two tabs: 'AL Utilities.codeunit.al' (active) and 'AL CustomerCard.pageext.al'. The main editor area shows the code for 'Utilities.codeunit.al'. The code is as follows:

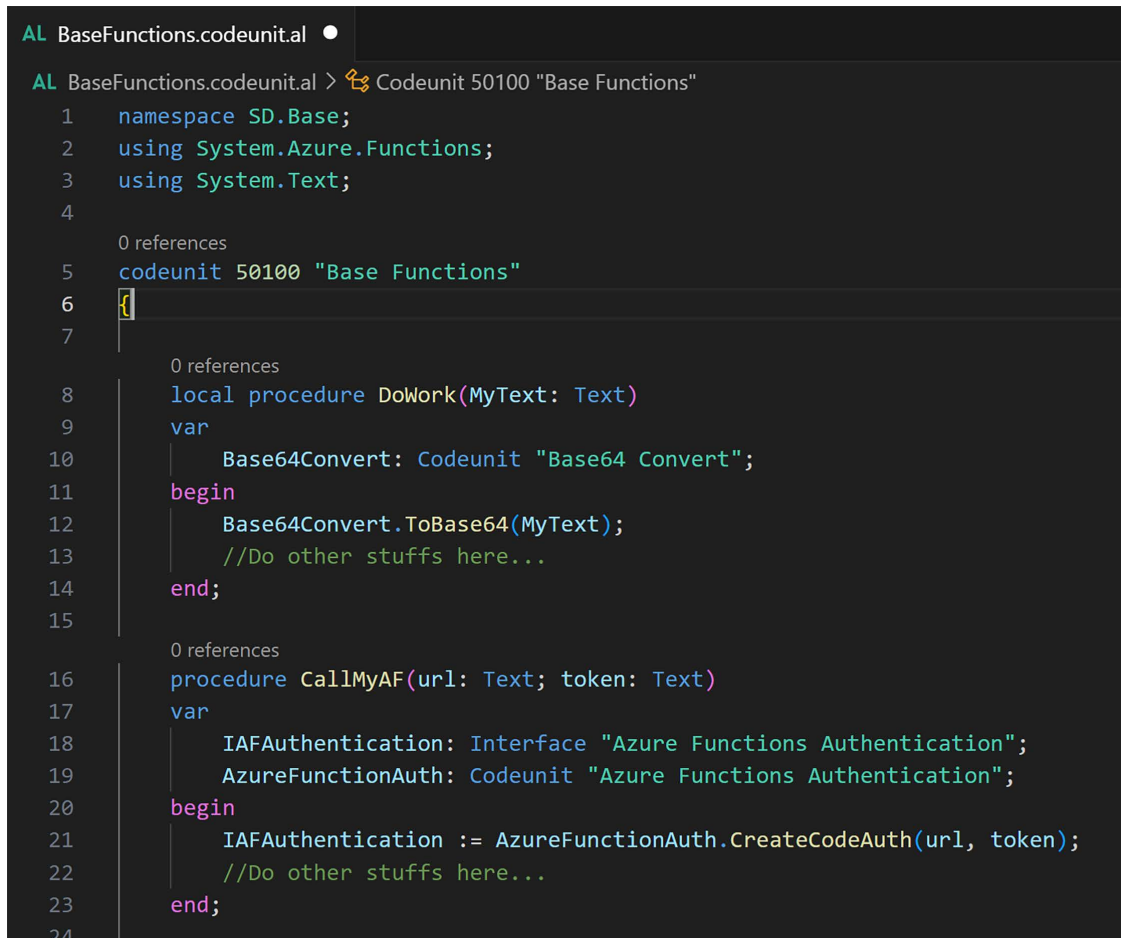
```
1 namespace SD.Utilities;
2
3   1 reference
4   codeunit 50101 Utilities
5   {
6     1 reference
7     procedure Utility1()
8     begin
9       //Do something here...
10    end;
11
12    0 references
13    procedure Utility2()
14    begin
15      //Do something here...
16    end;
17  }
```

The line 'namespace SD.Utilities;' on line 1 is highlighted with a red rectangular box. The code is color-coded: 'namespace' is blue, 'SD.Utilities;' is green, 'codeunit' is blue, '50101' is green, 'Utilities' is blue, 'procedure' is blue, 'Utility1()' is green, 'begin' is pink, 'end;' is pink, and 'Utility2()' is green.

Figure 6.18: Declaring an object belongs to a namespace

With the namespace keyword we define that the Utilities codeunit belongs to a namespace called SD.Utilities.

Then in our extension's code we can for example add a new codeunit that belongs to a new namespace, like in the following example:



```

AL BaseFunctions.codeunit.al •
AL BaseFunctions.codeunit.al > Codeunit 50100 "Base Functions"
1 namespace SD.Base;
2 using System.Azure.Functions;
3 using System.Text;
4
5 0 references
6 codeunit 50100 "Base Functions"
7 {
8   0 references
9   local procedure DoWork(MyText: Text)
10   var
11     Base64Convert: Codeunit "Base64 Convert";
12   begin
13     Base64Convert.ToBase64(MyText);
14     //Do other stuffs here...
15   end;
16
17   0 references
18   procedure CallMyAF(url: Text; token: Text)
19   var
20     IAFAuthentication: Interface "Azure Functions Authentication";
21     AzureFunctionAuth: Codeunit "Azure Functions Authentication";
22   begin
23     IAFAuthentication := AzureFunctionAuth.CreateCodeAuth(url, token);
24     //Do other stuffs here...
25   end;
26
27 }

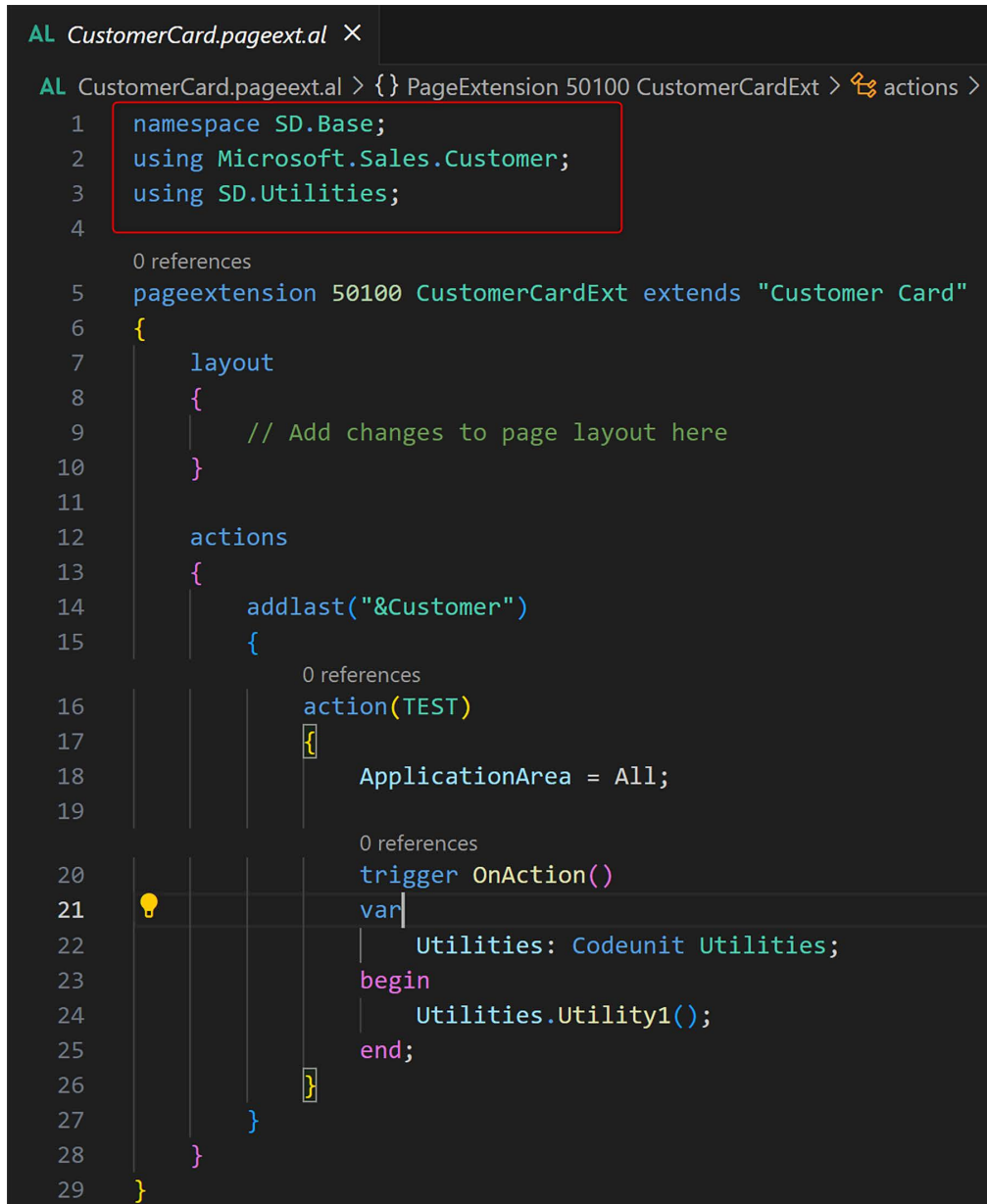
```

Figure 6.19: Adding a codeunit to a namespace

Here you can see that this new Base Functions codeunit belongs to the SD.Base namespace. Internally, this codeunit uses standard Dynamics 365 Business Central objects (like codeunit Base64 Convert, interface Azure Functions Authentication, and so on). All these objects belongs to a different namespace in the System application.

As you can see from the above image, now the AL language automatically adds a using clause to the object for including the external namespaces in use.

The same also applies also to other objects, like for example pageextension objects. If we want for example to extend the Customer Card page for adding an action to call the previously defined Utility1 procedure, we can define the pageextension object and define its namespace. Then when defining the action, the required namespaces to call the needed codeunit will be automatically added:



```

AL CustomerCard.pageext.al ×
AL CustomerCard.pageext.al > {} PageExtension 50100 CustomerCardExt > actions >
1 namespace SD.Base;
2 using Microsoft.Sales.Customer;
3 using SD.Utilities;
4
0 references
5 pageextension 50100 CustomerCardExt extends "Customer Card"
6 {
7     layout
8     {
9         // Add changes to page layout here
10    }
11
12    actions
13    {
14        addlast("&Customer")
15        {
16            0 references
17            action(TEST)
18            {
19                ApplicationArea = All;
20
21                0 references
22                trigger OnAction()
23                var
24                    Utilities: Codeunit Utilities;
25                begin
26                    Utilities.Utility1();
27                end;
28            }
29        }
30    }
31 }
  
```

Figure 6.20: Namespace added when defining an action

Please remember that:

- Renaming existing objects for using namespaces is a breaking change, so this capability mainly helps with the logical structure of existing objects and new names going forward.
- The AL language extension adds some code actions to help developers add namespaces to existing sources.

## Summary

In this chapter, we covered a lot of advanced topics and saw some tricks to implement particular tasks with the AL language extension.

We saw how to use access modifiers in code (useful to protect your object's visibility), how to use TryFunctions and collectible errors in order to create a better error-handling experience for your users, how to handle structured data with XML and JSON (useful to manage data coming from files or API or web service calls), and how to add security to extensions data by using Isolated Storage and Azure Key Vault.

We also saw how to improve the Dynamics 365 Business Central user experience by using control add-ins, and how to use asynchronous programming to execute tasks in the background without blocking the main UI (a useful trick to provide users with a fast page-opening experience).

You are now able to create complex extensions to improve the general user experience and handle different business tasks.

In the next chapter, we'll see how to customize, develop, and publish reports for Dynamics 365 Business Central.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>



# 7

## Handling Files with Dynamics 365 Business Central

Exchanging data via files is a standard practice in every company and every business. In a traditional application, you can read and write a file from the local file system easily. But what about a cloud-based application? Dynamics 365 Business Central is a SaaS application and you cannot directly interact with the local file system (you can if your extension targets the on-premises environment in the app.json file, but this is not “Universal Code-compliant”, as we learned in *Chapter 1, Dynamics 365 Business Central Momentum*).

How can (and should) you handle files with Dynamics 365 Business Central?

This chapter will cover the following topics:

- Using streams for loading files in the cloud
- Handling blobs in AL
- Using the Media and MediaSet data types for storing media files
- Using XMLPorts and CSV files
- Handling XML and JSON files from AL
- Understanding persistent blobs
- Using Isolated Storage for securing your data
- Using Azure Blob Storage from AL for storing files in an Azure storage account
- Using Azure file shares from AL

### Handling files with AL

Working with files is one of the trickier aspects of Dynamics 365 Business Central. While in the on-premises version, you have full access to your local resources and to a file system, in the SaaS version of Dynamics 365 Business Central things change. With the latter, you don't have a file system or access to local resources, since everything runs in Microsoft's datacenters.

If you create a procedure in AL, declaring a File variable and then invoking one of the common file management methods (like, for example, the Create method, which creates and opens an ASCII or binary file), this is the error that Visual Studio Code prompts you with:

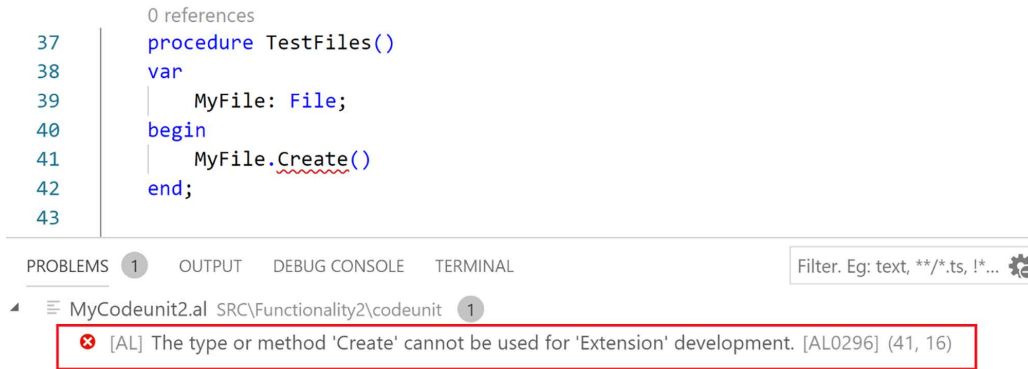


Figure 7.1: Error message in Visual Studio Code

This error occurs because the extension you're trying to create targets the Dynamics 365 Business Central SaaS environment by default. In the `app.json` file, you can see this in the code reading `"target": "Extension"`.

If you change the `app.json` code to `"target": "Internal"`, you're declaring that your extension is for the on-premises world only. The error disappears and you can use the classic File methods:

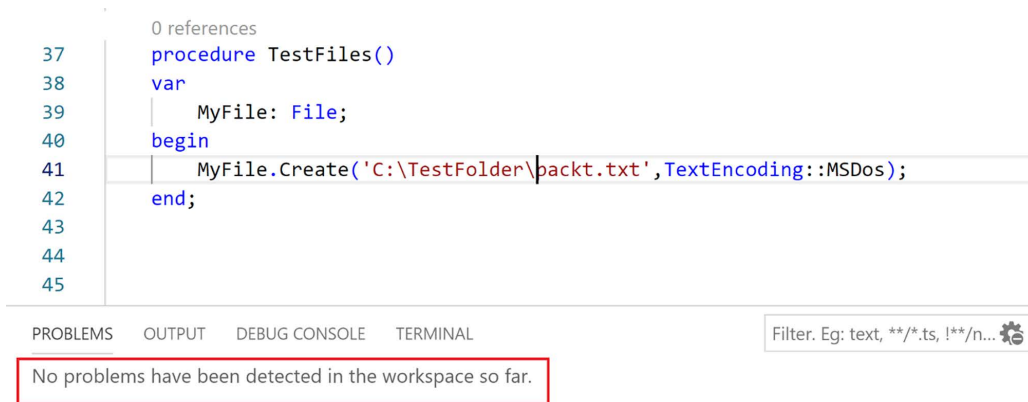


Figure 7.2: Error message removed

The same happens if you use the FileManagement codeunit for handling files:



Figure 7.3: Error message for the FileManagement codeunit

So, this allows you to create extensions that work on-premises. However, to handle files in the cloud environment, you need to use **streams**. Streams are an important concept that we'll go on to use later in this chapter. Chiefly, we're looking at the InStream and OutStream objects.

The InStream and OutStream data types are generic stream objects used for reading from or writing to files and blobs. More information about these objects can be found at the following links:

- <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/instream/instream-data-type>
- <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/outstream/outstream-data-type>

Now, let's look at some methods using these stream objects.

To upload a file from the client computer to a server-side stream object interactively (by opening a dialog to the user), you need to call the UploadIntoStream method:

```
[Ok := ] File.UploadIntoStream(DialogTitle: String, FromFolder: String,
    FromFilter: String, var FromFile: Text, var InStream: InStream)
```

The method's parameters are the following:

- **DialogTitle (String):** The text that will be shown in the title bar of the dialog box, expressed as a string.
- **FromFolder (String):** This string contains the path to the folder displayed in the dialog box. This is the default folder, but the user can browse to any available location.
- **FromFilter (String):** A string specifying the type of files that can be uploaded to the server side. In the Windows client, the type is displayed in the upload dialog box, so the user can only select files of the specified type. The user can try to upload any file type, but an error occurs if the file is not the specified type.
- **FromFile (Text):** The default file to upload to the service. The user can change the file.
- **InStream:** The InStream object to load data in.

More details about the UploadIntoStream method, such as web client support, can be found here:

- <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/file/file-uploadintostream-string-instream-method>
- <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/file/file-uploadintostream-string-string-string-text-instream-method>

Let's apply this method with an example. The following function adds an picture item. It does this by using UploadIntoStream to load the client file into InStream, and then loads the image file to the client side using InStream:

```
procedure ImportItemPicture(Item: Record Item)
var
    FileInStream: InStream;
    FileName: Text;
begin
    if UploadIntoStream('', '', '', FileName, FileInStream) then
        begin
            Clear(Item.Picture);
            Item.Picture.ImportStream(FileInStream, FileName);
            Item.Modify(true);
        end;
end;
```

As another example, this is a function that reads a CSV file that contains the Item details into an InStream object, loads its content into the CSV Buffer table, and then updates the Item fields accordingly:

```
local procedure UploadCSV()
var
    CSVInStream : InStream;
```

```

UploadResult : Boolean;
TempBlob : Codeunit "Temp Blob";
DialogCaption, CSVFileName : Text;
CSVBuffer: Record "CSV Buffer";
Item: Record Item;
begin
    UploadResult := UploadIntoStream(DialogCaption,
,CSVFileName,CSVInStream);
    CSVBuffer.DeleteAll;
    CSVBuffer.LoadDataFromStream(CSVInStream,';');
    if CSVBuffer.FindSet() then
        repeat
            if (CSVBuffer."Field No." = 1) then
                Item.Init();
                case CSVBuffer."Field No." of
                    1: Item.Validate("No." '','',CSVBuffer.Value);
                    2: Item.Validate("Description",CSVBuffer.Value);
                    3: Item.Validate("Item Category Code",CSVBuffer.Value);
                    4: if not Item.Insert() then Item.Modify();
                end;
            until CSVBuffer.Next()=0;
        end;
    end;
end;

```

Now, let's move on to another method. To download a file from the server side (SaaS) to the client side, you need to use the DownloadFromStream method:

```

[Ok := ] File.DownloadFromStream(InStream: InStream, DialogTitle: String,
ToFolder: String, ToFilter: String, var ToFile: Text)

```

The method's parameters are the following:

- InStream: The InStream object containing the data.
- DialogTitle (String): This string should contain the title that will be displayed in the dialog box for downloading the file.
- ToFolder (String): The default folder where the downloaded file will be saved, expressed as a string. The folder name is shown in the dialog box for downloading. The user can change this value.
- ToFilter (String): A string that specifies the types of files that can be downloaded to the client. The type is displayed in the dialog box for downloading the file.
- ToFile (Text): The name assigned to the downloaded file. This value can be changed by the user.

More information about the DownloadFromStream method, including web client compatibility, can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/file/file-downloadfromstream-method>.

Again, let's try the method using an example. The following code exports the images associated with an Item card as the MediaSet type. It uses DownloadFromStream to download the file to the client. The images are retrieved from the Tenant Media table and saved with a file name that is the item number, plus the image index and image extension. For this, there's a GetImageExtension function to retrieve the image file extension according to its MIME type:

```

procedure ExportItemPicture(Item: Record Item)
var
    FileInStream: InStream;
    FileName: Text;
    i: Integer;
    TenantMedia: Record "Tenant Media";
    ErrMsg: Label 'No images stored for the selected item.';
begin
    if Item.Picture.Count() = 0 then
        Error(ErrMsg);

        for i := 1 to Item.Picture.Count() do begin
            if TenantMedia.Get(Item.Picture.MediaId()
        ) then begin
            TenantMedia.CalcFields(Content);
            if TenantMedia.Content.HasValue() then begin
                FileName := Item."No." + '_' + Format(i) +
GetImageExtension(TenantMedia);
                TenantMedia.Content.CreateInStream(FileInStream);
                DownloadFromStream(FileInStream, '', '', '', FileName);
            end;
        end;
    end;
end;

procedure GetImageExtension(var TenantMedia: record "Tenant Media"): Text
begin
    case TenantMedia."Mime Type" of
        'image/jpeg': exit('.jpg');
        'image/bmp': exit('.bmp');
        'image/png': exit('.png');
        'image/gif': exit('.gif');
        'image/tiff': exit('.tiff');
        'image/wmf': exit('.wmf');
    end
end;

```

If you need to create a file from Dynamics 365 Business Central, it cannot be done directly in the file system. Instead, you need to create it on the server side using the Temp Blob codeunit and the OutStream object. Then, you need to download it to the client by using an InStream object. We'll learn about blob code later in this chapter.

We haven't covered blobs yet, but let's look at an example using this method, so we can see how InStream works. This is an example of an AL function that receives a file name as input, then creates a text file with three lines, and downloads it to the client side:

```
procedure CreateTextFile(FileName: Text)
var
    InStr: InStream;
    OutStr: OutStream;
    TempBlob: Codeunit "Temp Blob";
    CR, LF: char;
begin
    CR := 13;
    LF := 10;
    TempBlob.CreateOutStream(OutStr);
    OutStr.WriteText('First line' + CR + LF);
    OutStr.WriteText('Second line' + CR + LF);
    OutStr.WriteText('Third line' + CR + LF);
    TempBlob.CreateInStream(InStr);
    DownloadFromStream(InStr, '', '', '', FileName);
end;
```

Here you can see how the TempBlob codeunit creates the file, which the stream objects then work on by writing text. How the blob code works may make more sense once we work on blobs in the *Using Persistent Blobs* and *Using Azure Blob Storage from AL* sections.

Learning how to use streams is an essential skill for handling files in Business Central in the age of cloud computing. In our examples, we've looked at text and image files. We recommend you try using streams in more ways, so you can identify how to use them in your work as a Business Central developer.

## Handling attachments

Attachments are files that you can link to entities or documents in Dynamics 365 Business Central. There are two main tables used to store attachments:

- Document Attachment (ID = 1173)
- Attachment (ID = 5062)

To store an attachment in these tables and then to download it from these tables you need to use the aforementioned UploadIntoStream and DownloadFromStream methods.

An example function to upload a file to the Attachment table is as follows:

```

procedure UploadAttachment()
    var
        Attachment: Record Attachment;
        outStr: OutStream;
        inStr: InStream;
        tempfilename: text;
        FileMgt: Codeunit "File Management";
        DialogTitle: Label 'Please select a File...';
    begin
        if UploadIntoStream(DialogTitle, '', 'All Files (*.*)|*.*',
            tempfilename, inStr) then begin

            Attachment.Init();
            Attachment.Insert(true);
            Attachment."Storage Type" := Attachment."Storage Type"::Embedded;
            Attachment."Storage Pointer" := '';
            Attachment."File Extension" := FileMgt.GetExtension(tempfilename);
            Attachment."Attachment File".CreateOutStream(outStr);
            CopyStream(outStr, inStr);
            Attachment.Modify(true);

        end;
    end;
end;

```

An example function to download a file from the Attachment table is as follows:

```

procedure OpenAttachment(AttachmentEntryNo: Integer)
    var
        Attachment: Record Attachment;
        inStr: InStream;
        tempfilename: Text;
        ErrorAttachment: Label 'File not available.';
    begin
        if Attachment.get(AttachmentEntryNo) then
            if Attachment."Attachment File".HasValue then begin
                Attachment.CalcFields("Attachment File");
                Attachment."Attachment File".CreateInStream(inStr);
                tempfilename := CreateGuid() + '.' + Attachment."File
Extension";
                DOWNLOADFROMSTREAM(inStr, 'Save file', '', 'All Files
(*.*)|*.*', tempfilename);
            end;
        end;
    end;
end;

```

```
        end
        else
            Error(ErrorAttachment);
        end;
end;
```

The same applies to the Document Attachment table, where you only need to add the reference to the document itself to the attachment record.

## Reading and writing text data from blob fields

To read and write text data to and from a blob field, you need to use `InStream` and `OutStream` objects as previously described. The following two methods read and write text data to a blob field defined in a custom table:

```
table 50120 MyBlobTable
{
    DataClassification = CustomerContent;

    fields
    {
        field(1; ID; Integer)
        {
            DataClassification = CustomerContent;
        }
        field(2; BlobField; Blob)
        {
            DataClassification = CustomerContent;
        }
    }

    keys
    {
        key(PK; ID)
        {
            Clustered = true;
        }
    }

    procedure SetBlobValue(value: Text)
    var
        outStr: OutStream;
    begin
        BlobField.CreateOutStream(outStr);
```

```

        outStr.WriteText(value);
    end;

    procedure GetBlobValue(value: Text)
    var
        inStr: InStream;
    begin
        CalcFields(BlobField);
        if BlobField.HasValue() then
            begin
                BlobField.CreateInStream(inStr);
                inStr.ReadText(value);
            end
        else
            value := 'No value on the BLOB field';
        end;
    }

```

## Using the Media and MediaSet data types in AL code

In Dynamics 365 Business Central you have two possible ways to store media contents (like images, PDF files, etc.) inside the database: using a blob field (as explained in the previous section) or using the **Media** and **MediaSet** data types.

With the **Media** and **MediaSet** data types you can store media in system tables of the database, and then reference the media from your records in the application.

Using the **Media** and **MediaSet** data types gives you better performances than using a blob data type and also it guarantees a more flexible design. With the blob data type, each time the media is rendered in the client, it's retrieved from the SQL database, which requires extra bandwidth and affects performance. With the **Media** and **MediaSet** data types, the client uses the media ID to cache the media data, which in turn improves the response time for rendering the media in the user interface.

The **Media** data type is used to save a single media file. Imported media is stored as an object in the system table **2000000184 Tenant Media** of the tenant database. Each media object is assigned a unique identifier (ID).

The **MediaSet** data type instead associates a record with one or more media objects. If a media object is added to a **MediaSet** data type field, the media object is assigned to a media set in the system table **2000000183 Tenant Media Set**. A unique identifier is assigned to this media set, which is then referenced from the field. The media set is created with the first file media object that you add to the record. Any other media objects for the record are then associated with the same media set.

To load the content of a Media field you can use the `ImportStream` method:

```
[ID := ] Media.ImportStream(Stream: InStream, Description: Text, MimeType:
Text, FileName: Text)

local procedure LoadMedia()
var
    Item: Record Item;
    Instr: InStream;
    fromFile: Text;
begin
    if UploadIntoStream('Import Media', '', 'All files(*.*)|*.*', fromFile,
Instr ) then
        begin
            Clear(Rec.Image);
            Rec.Image.ImportStream(Instr, fromFile);
            Rec.Modify(true);
        end;
    end;
end;
```



When using `MediaSet` data type, Microsoft alerts you when a table record that contains a Media object is deleted; the `OnDelete` trigger gets the Media or MediaSet's ID. It then uses the ID to look for other references to the Media object from the same field index in the same table. If no other references are found, the Media object is assumed to be unreferenced and it's deleted.

Note that the runtime won't look in all tables in the database to see if a Media object is referenced elsewhere, since doing so would lower performance with costly SQL table scans.

To avoid this problem, Microsoft introduced a new `FindOrphans` method to the Media object, which is useful for discovering all orphaned media (orphaned media is media that is not referenced by any other table):

```
procedure FindOrphans(): List of [Guid]
```

You can use this method to delete all orphaned media and free the allocated space in your tenant in the following way:

```
local procedure FindMediaOrphans()
var
    TenantMedia: Record "Tenant Media";
    MediaOrphans: List of [Guid];
    MediaId: Guid;
begin
```

```
MediaOrphans := Media.FindOrphans();
foreach MediaId in MediaOrphans do begin
    if TenantMedia.Get(MediaId) then
        TenantMedia.Delete();
    end;
end;
```

## Using XMLport in AL code

**XMLport** objects are used to import and export data between Dynamics 365 Business Central and external data sources (this is managed by the **Direction** property that can be set to **Import**, **Export**, **Both**). Data can be imported from or exported to as **XML** or **CSV** (text) format (the **Format** property can be set to **Xml**, **Variable Text**, or **Fixed Text**).

The XMLport object properties are detailed here: <https://docs.microsoft.com/en-us/dynamics-nav/xmlport-properties>.

The XMLport object triggers are detailed here: <https://docs.microsoft.com/en-us/dynamics-nav/xmlport-triggers>.

Now consider the sample XMLport object defined in Chapter 3:

```
xmlport 50100 MyXmlportImportCustomer
{
    Direction = Import;
    Format = VariableText;
    FieldSeparator = ';';
    RecordSeparator = '<LF>';
    schema
    {
        textelement(NodeName1)
        {
            tableelement(Customer; Customer)
            {
                fieldattribute(No; Customer."No.")
                {
                }
                fieldattribute(Name; Customer.Name)
                {
                }
                fieldattribute(Address; Customer.Address)
                {
                }
            }
        }
    }
}
```

```

    }
    fieldattribute(City;Customer.City)
    {
    }
    fieldattribute(Country;Customer."Country/Region Code")
    {
        trigger OnAfterAssignField()
        begin
            //Executed after a field has been assigned a value and
            before it is validated and imported.
        end;
    }
}
}
}
}
}

```

To execute an XMLport object in Dynamics 365 Business Central, you need to run it from a page or a codeunit object (you cannot directly run it). XMLport request pages (used to set filters or insert parameters) in the Dynamics 365 Business Central web client are not supported.

To execute an XMLport object in Dynamics 365 Business Central for importing data from a file, you need to use the following code:

```

procedure RunXMLportImport()
var
    FileInstream: InStream;
    FileName: Text;
begin
    UploadIntoStream('', '', '', FileName, FileInstream);
    Xmlport.Import(Xmlport::MyXmlportImportCustomer, FileInstream);
    Message('Import Done successfully.');
```

Here the file is loaded into an InStream object and then the XMLport object is executed by passing the InStream object as input.

To execute an XMLport object in Dynamics 365 Business Central for exporting data to a file, you need to use the following code:

```

procedure RunXMLportExport()
var
    TempBlob: Codeunit "Temp Blob";
```

```

    FileName, outputFileName: Text;
    FileOutputStream: OutStream;
    FileInStream: InStream;
begin
    TempBlob.CREATEOUTSTREAM(FileOutputStream);
    Xmlport.Export(Xmlport::MyXmlportImportCustomer, FileOutputStream);
    TempBlob.CREATEINSTREAM(FileInStream);
    outputFileName := 'MyOutputFile.xml';
    DownloadFromStream('','',FileInStream, outputFileName);
    //The output is saved in the default browser's Download folder
end;

```

## Handling XML and JSON files with AL

The AL language extension has native support for handling XML and JSON documents.

An XML document is represented by using the **XmlDocument** data type: <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/xml/xmldocument/xmldocument-data-type>.

The following code shows how you can import an XML file and load it into an XmlDocument object:

```

local procedure ImportXML()
var
    TempBlob : Codeunit "Temp Blob";
    TargetXmlDoc : XmlDocument;
    XmlDec : XmlDeclaration;
    Instr: InStream;
    filename: Text;
begin
    // Create the Xml Document
    TargetXmlDoc := XmlDocument.Create;
xmlDec := xmlDeclaration.Create('1.0','UTF-8','');
    TargetXmlDoc.SetDeclaration(xmlDec);
    // Create an Instream object & upload the XML file into it
    Tempblob.CreateInStream(Instr);
    filename := 'data.xml';
    UploadIntoStream('Import XML',' ',filename,Instr);

    // Read stream into new xml document
    XmlDocument.ReadFrom(Instr, TargetXmlDoc);
end;

```

Here, we create an `XmlDocument` object with an XML declaration, then create an `InStream` object for loading the XML file, and then we read the `InStream` content into the `XmlDocument` object.

If you reference the `TargetXmlDoc` object, you have all the available methods for handling and manipulating the XML file:

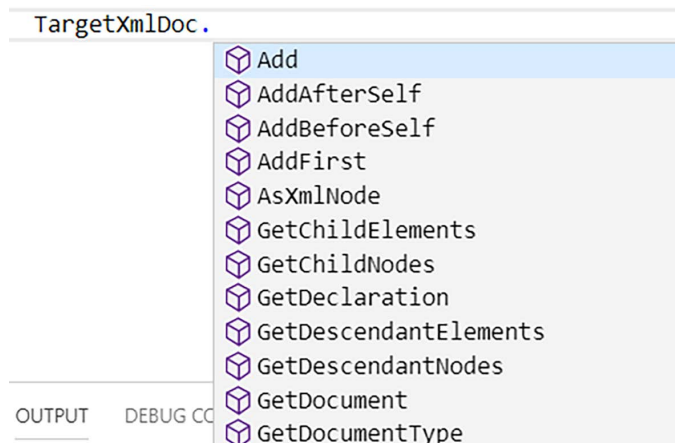


Figure 7.4: List of methods for XML file handling

To create an XML document directly from AL code, you can use the `XmlDocument` and `XmlElement` classes:

```
local procedure XMLDocumentCreation()
var
    xmlDoc: XmlDocument;
    xmlDec: XmlDeclaration;
    node1, node2: XmlElement;
    node: XmlNode;
begin
    xmlDoc := XmlDocument.Create();
    xmlDec := xmlDeclaration.Create('1.0', 'UTF-8', '');
    xmlDoc.SetDeclaration(xmlDec);

    node1 := XmlElement.Create('node1');
    xmlDoc.Add(node1);
    node2 := XmlElement.Create('node2');
    node2.SetAttribute('ID', '3');
    node1.Add(node2);
end;
```

This code creates an XML document with a root node (called node1) and a child node (called node2) with an ID attribute with a value of 3 ( <node2 ID="3"> ).

The native support for JSON documents is given by using the **JsonObject** and **JsonArray** data types. Each of these data types contains the methods for handling a JSON file (read and write) and for manipulating the JSON data (tokens). A detailed explanation of all the available methods for these data types can be found at the following links:

- <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/jsonobject/jsonobject-data-type>
- <https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/jsonarray/jsonarray-data-type>

The following code shows an example of how to create a JSON representation of a Sales Order document:

```
procedure CreateJsonOrder(OrderNo: Code[20])
var
    JsonObjectHeader: JsonObject;
    JsonObjectLines: JsonObject;
    JsonOrderArray: JsonArray;
    JsonArrayLines: JsonArray;
    SalesHeader: Record "Sales Header";
    SalesLines: Record "Sales Line";
begin

    //Retrieves the Sales Header
    SalesHeader.Get(SalesHeader."Document Type"::Order, OrderNo);
    //Creates the JSON header details
    JsonObjectHeader.Add('sales_order_no', SalesHeader."No.");
    JsonObjectHeader.Add(' bill_to_customer_no', SalesHeader."Bill-to
Customer No.");
    JsonObjectHeader.Add('bill_to_name', SalesHeader."Bill-to Name");
    JsonObjectHeader.Add('order_date', SalesHeader."Order Date");
    JsonOrderArray.Add(JsonObjectHeader);

    //Retrieves the Sales Lines
    SalesLines.SetRange("Document Type", SalesLines."Document
Type"::Order);
    SalesLines.SetRange("Document No.", SalesHeader."No.");
    if SalesLines.FindSet then

        // JsonObject Init
        JsonObjectLines.Add('line_no', '');
```

```

        JsonObjectLines.Add('item_no', '');
        JsonObjectLines.Add('description', '');
        JsonObjectLines.Add('location_code', '');
        JsonObjectLines.Add('quantity', '');

        repeat
            JsonObjectLines.Replace('line_no', SalesLines."Line No.");
            JsonObjectLines.Replace('item_no', SalesLines."No.");
        JsonObjectLines.Replace('description', SalesLines.Description);
            JsonObjectLines.Replace('location_code', SalesLines."Location
Code");
            JsonObjectLines.Replace('quantity', SalesLines.Quantity);
            JsonObjectLines.Add(JsonObjectLines);
        until SalesLines.Next() = 0;

        JsonObjectArray.Add(JsonObjectLines);

    end;

```

The procedure receives an order number as input, retrieves the Sales Header and Sales Line details, and creates a JSON representation. The final result is as follows:

```

[
  {
    "sales_order_no": "S01900027",
    "bill_to_customer_no": "C001435",
    "bill_to_name": "Packt Publishing",
    "order_date": "2019-03-23"
  },
  [
    {
      "line_no": "10000",
      "item_no": "IT00256",
      "description": "Dynamics 365 Business Central Development
Guide",
      "location_code": "MAIN",
      "quantity": 30
    },
    {
      "line_no": "20000",
      "item_no": "IT03465",
      "description": "Mastering Dynamics 365 Business Central",
      "location_code": "MAIN",

```

```

        "quantity": 27
    }
]
]

```

Obviously, you can also receive a JSON representation as input (for example, as a response from an API call) and handle it using the same data types.

## Using persistent blobs

Sometimes when creating applications in Dynamics 365 Business Central there's the need to handle blob fields linked to an entity. Creating a blob field in an entity table is something that you should always carefully evaluate if needed. Sometimes it's better to store the file elsewhere and only use a reference to the original entity.

If you check Microsoft's base application, you will discover that a table called `Persistent Blob` is available (table Id = 4151).

This table is not directly accessible by code (its access level is `Internal`) and it's defined as follows:

```

table 4151 "Persistent Blob"
{
    Access = Internal;

    fields
    {
        field(1; "Primary Key"; BigInteger)
        {
            AutoIncrement = true;
            DataClassification = SystemMetadata;
        }
        field(2; Blob; BLOB)
        {
            DataClassification = CustomerContent;
        }
    }

    keys
    {
        key(Key1; "Primary Key")
        {
            Clustered = true;
        }
    }
}

```

```

    }
}

fieldgroups
{
}
}

```

The idea behind persistent blobs is that the link to a file is not a local path but rather a key to the **Persistent Blob** table record, by which the required file is found. The key must be stored in the selected table/tables for later use to find the file you are looking for.

This table is used to store blob values between sessions and you can use it via the **Persistent Blob** codeunit, which exposes the following methods:

- Create
- Exists
- Delete
- CopyFromInStream
- CopyToOutStream

The following code shows how you can import a file into the **Persistent Blob** table:

```

local procedure ImportPersistentBlob(var BlobKey: BigInteger; var FileName:
Text)
var
    PersistentBlob: Codeunit "Persistent Blob";
    FileInstream: InStream;
begin
    if UploadIntoStream('Upload File', '', '', FileName, FileInstream) then
    begin
        BlobKey := PersistentBlob.Create();
        PersistentBlob.CopyFromInStream(BlobKey, FileInstream);
    end;
end;

```

The following code shows you how you can export a blob from the **Persistent Blob** table:

```

local procedure ExportPersistentBlob(BlobKey: BigInteger; FileName: Text)
var
    PersistentBlob: Codeunit "Persistent Blob";
    TempBlob: Codeunit "Temp Blob";
    FileOutstream: OutStream;
    FileInstream: InStream;
begin

```

```
if PersistentBlob.Exists(BlobKey) then begin
    TempBlob.CreateOutputStream(FileOutputStream);
    PersistentBlob.CopyToOutputStream(BlobKey, FileOutputStream);
    TempBlob.CreateInStream(FileInStream);

    DownloadFromStream(FileInStream, , '', '', '' FileName);
end else
    Error('Persistent Blob does not exist');
end;
```

## Isolated Storage

**Isolated Storage** is a key-value-based storage that provides data isolation between extensions. Isolated Storage can be used to store data that must be preserved inside the extension scope and is accessible via AL code. Note that, when working with Isolated Storage, we are working with key values that we can access without having to write to the database; it's a way to store persistent information, not files.

The `DataScope` option type identifies the scope of stored data in Isolated Storage.

`DataScope` is an optional parameter and *if it is not passed*, the value will be `Module`. Possible values are listed in the following table:

Member	Description
Module	Indicates that the record is available in the scope of the app (extension) context.
Company	Indicates that the record is available in the scope of the company within the app context.
User	Indicates that the record is available for a user within the app context.
CompanyAndUser	Indicates that the record is available for a user and specific company within the app context.

Table 7.1: *DataScope* values

For managing data in Isolated Storage, you have the following methods:

Method	Description
<pre>[Ok := ] IsolatedStorage. Set(Key: String, Value: String, [DataScope: DataScope])</pre>	Sets the value associated with the specified key within the extension. The optional <code>DataScope</code> parameter is the scope of the stored data.
<pre>[Ok := ] IsolatedStorage. Get(Key: String, [DataScope: DataScope], var Value: Text)</pre>	Gets the value associated with the specified key within the extension. The optional <code>DataScope</code> parameter is the scope of the data to retrieve.

<pre>HasValue := IsolatedStorage. Contains(Key: String, [DataScope: DataScope])</pre>	Determines whether the storage contains a value with the specified key within the extension. The optional DataScope parameter is the scope in which to check for the existence of a value with the given key.
<pre>[Ok := ] IsolatedStorage. Delete(Key: String, [DataScope: DataScope])</pre>	Deletes the value with the specified key from Isolated Storage within the extension. The optional DataScope parameter is the scope from which to remove the value with the given key.

Table 7.2: Isolated Storage methods

Isolated Storage is useful for storing sensitive data, user options, and license keys.

Let’s consider the following example:

```
local procedure IsolatedStorageTest()  
  var  
    keyValue: Text;  
  begin  
    IsolatedStorage.Set('mykey', 'myvalue', DataScope::Company);  
  
    if IsolatedStorage.Contains('mykey', DataScope::Company) then  
      begin  
        IsolatedStorage.Get('mykey', DataScope::Company, keyValue);  
        Message('Key value retrieved is %1', keyValue);  
      end;  
  
      IsolatedStorage.Delete('mykey', DataScope::Company);  
    end;
```

As the first step in the preceding code, we save in Isolated Storage a key called mykey with a value of myvalue and DataScope = Company. The key is visible in the scope of the company within the app context, so no other extensions can access this key.

In the second step, we check if a key called mykey is saved in Isolated Storage with DataScope = Company. If the match is found (key and scope) the key is retrieved (with the Get method) and the value is returned in the keyValue text variable.

In the last step, we delete the key for this DataScope.

As previously said, you could use Isolated Storage also for saving license keys or license details for your extension. The following code shows how to export the records of a table called License to JSON, then how to encrypt the JSON value, and finally how to store the encrypted text in Isolated Storage:

```
local procedure StoreLicense()  
  var  
    StorageKey: Text;
```

```

    LicenseText: Text;
    EncryptManagement: Codeunit "Encryption Management";
    License: Record License temporary;
begin
    StorageKey := GetStorageKey();
    LicenseText := License.WriteLicenseToJson();
    if EncryptManagement.IsEncryptionEnabled() and EncryptManagement.
IsEncryptionPossible() then
        LicenseText := EncryptManagement.Encrypt(LicenseText);

    if IsolatedStorage.Contains(StorageKey, DataScope::Module) then
        IsolatedStorage.Delete(StorageKey);

    IsolatedStorage.Set(StorageKey, LicenseText, DataScope::Module);
end;

local procedure GetStorageKey(): Text
var
    //Returns a GUID
    StorageKeyTxt: Label 'dd03d28e-4acb-48d9-9520-c854495362b6', Locked =
true;
begin
    exit(StorageKeyTxt);
end;

local procedure ReadLicense()
var
    StorageKey: Text;
    LicenseText: Text;
    EncryptManagement: Codeunit "Encryption Management";
    License: Record License temporary;
begin
    StorageKey := GetStorageKey();
    if IsolatedStorage.Contains(StorageKey, DataScope::Module) then
        IsolatedStorage.Get(StorageKey, DataScope::Module, LicenseText);

    if EncryptManagement.IsEncryptionEnabled() and EncryptManagement.
IsEncryptionPossible() then
        LicenseText := EncryptManagement.Decrypt(LicenseText);

    License.ReadLicenseFromJson(LicenseText);
end;

```

Here, the `License` table is declared as a temporary table. In this way, the data is isolated into the calling codeunit.

Please remember the following:

- In Dynamics 365 Business Central SaaS, sensitive data stored in Isolated Storage is always encrypted.
- In Dynamics 365 Business Central on-premises, encryption is controlled by the end user (via the *Data Encryption Management* page), and here:
  - If encryption is turned on, a secret stored in Isolated Storage is automatically encrypted.
  - A secret that was inserted while encryption was turned off will remain unencrypted even after encryption is turned on.
  - If you turn off encryption, the secret will be decrypted.

As a result, if you have an extension that works for Dynamics 365 Business Central SaaS and on-premises (same code) and you're using Isolated Storage to store secrets, you need to check if the encryption is enabled (always true for SaaS) and then save the secret accordingly.

So, a function that saves a license key to Isolated Storage and that works for Dynamics 365 Business Central SaaS and on-premises will be as follows:

```
local procedure SaveLicense()
var
    licenseKeyValue: Text;
begin
    if not EncryptionEnabled() then
        IsolatedStorage.Set('LicenseKey', licenseKeyValue, DataScope::Module)
    else
        IsolatedStorage.
SetEncrypted('LicenseKey', licenseKeyValue, DataScope::Module)
end;
```

## Using Azure Blob Storage from AL

In a SaaS environment, **storage capacity** is something that must be considered when designing Dynamics 365 Business Central solutions.

By default, Dynamics 365 Business Central customers can use up to 80 GB of database storage capacity across all their environments (production and sandbox). The Premium and Essential license types give each Dynamics 365 Business Central customer one production environment and three sandbox environments free of charge. If the customer requires more production environments, they can buy additional environments through their CSP partner. Each additional production environment comes with three additional sandbox environments and 4 GB of additional tenant-wide database capacity.

Customers also benefit from additional storage capacity based on the number of Business Central licenses they own:

- **Premium** license: 3 GB of additional storage for each license.
- **Essential** license: 2 GB of additional storage for each license.
- **Device** license: 1 GB of additional storage for each license.

Customers can then purchase additional database capacity and environments through their reselling partner by using the following add-ons to their existing license:

- Dynamics 365 Business Central Database Capacity (1 GB)
- Dynamics 365 Business Central Database Capacity (100 GB)
- Dynamics 365 Business Central Database Capacity Overage (1 GB; a lower-priced add-on is only available for the customers who purchased at least one Dynamics 365 Business Central Database Capacity (100 GB) add-on)
- Dynamics 365 Business Central Additional Environment add-on

As you can see, storage carries a cost in cloud environments and you should bear that in mind from the very start when creating solutions. When approaching the handling of files, a common mistake in a Dynamics 365 Business Central cloud environment is to store blobs in the database itself. This is not optimal and saving a lot of files directly to the database increases its storage size.

Dynamics 365 Business Central can interact with Azure Blob Storage to handle files in the cloud without using the database.

**Azure Blob Storage** is a cloud storage service optimized for storing massive amounts of unstructured data such as text, images, and binary data in general.

Blob Storage in Azure includes three types of resources:

- **Storage account:** The unique namespace for your storage data
- **Container:** An organized set of blobs of the same type
- **Blob:** The physical blob object saved on a container

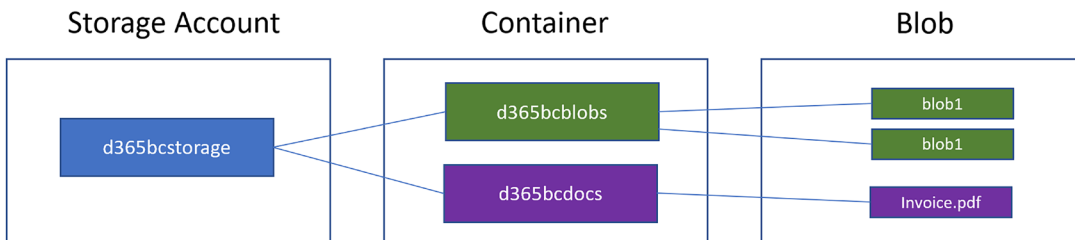


Figure 7.5: Azure Blob Storage resources

In the following sections, we'll see how to configure Azure Blob Storage and how to use it via AL code. The code for the example we'll use can be found in the book's GitHub repo (see the "Azure Blob Storage Attachments" extension): <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition/tree/main/CH07/Azure%20Blob%20Storage%20Attachments>.

# Creating a storage account in Azure

To create a storage account in Azure, open the Azure portal and search for **Storage Account**. Then create a new Storage Account instance by selecting the **Azure subscription**, the **Resource group** (recommended to create a dedicated resource group), the **storage account name**, and the **Azure region** on which the storage account will be created.

When creating a new storage account you need also to select the performance tier (Standard or Premium) and the redundancy of the storage. For handling files with Dynamics 365 Business Central I suggest starting with the low-cost option of the Standard tier with **locally redundant storage (LRS)** or **geo-redundant storage (GRS)**:

Home > Storage accounts >

## Create a storage account

Basics | Advanced | Networking | Data protection | Encryption | Tags | Review

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. [Learn more about Azure storage accounts](#)

### Project details

Select the subscription in which to create the new storage account. Choose a new or existing resource group to organize and manage your storage account together with other resources.

Subscription \*

Resource group \*

demostraregery

Create new

### Instance details

If you need to create a legacy storage account type, please click [here](#).

Storage account name ⓘ \*

Region ⓘ \*

Performance ⓘ \*

Redundancy ⓘ \*

d365bcblobsd

(Europe) West Europe

☒ Standard: Recommended for most scenarios (general-purpose v2 account)

☐ Premium: Recommended for scenarios that require low latency.

Locally-redundant storage (LRS)

Deploy to an edge zone

Figure 7.6: Provisioning a storage account

When the storage account is provisioned, to use it from Dynamics 365 Business Central you need to retrieve its access key. To do that, from the newly created storage account, select the **Access Key** blade and copy the **key1** or **key2** value:

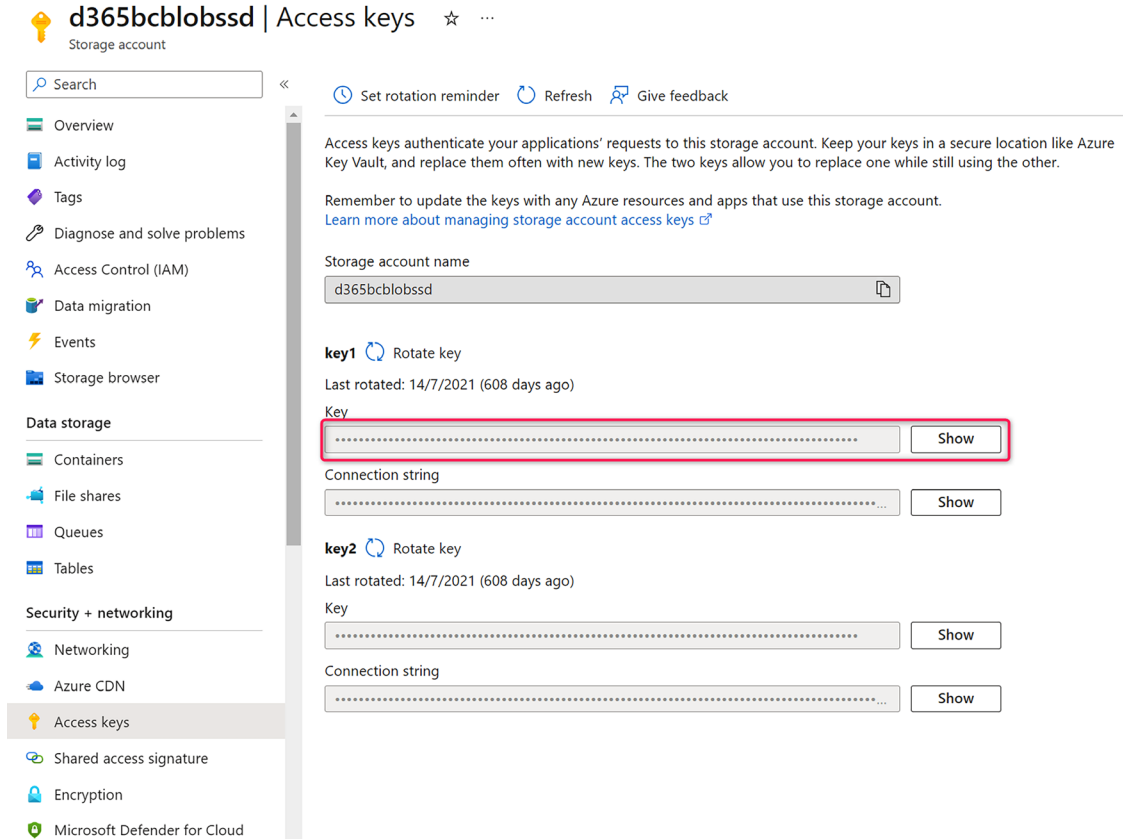


Figure 7.7: Copying access key values

You're now ready to use it from AL code.

## Using Azure Blob Storage from AL code

Dynamics 365 Business Central has a built-in module in the **System Application** that permits you to interact with Azure Blob Storage. With the provided objects you can authenticate to the storage account, create containers or list existing containers, create, list, and delete blobs, and so on.

The following code shows how to handle a series of common operations with the Azure Blob Storage from AL code:

```
codeunit 50101 BlobStorageManagement
{
    trigger OnRun()
    var
```

```

ContainerClient: Codeunit "ABS Container Client";
StorageServiceAuthorization: Codeunit "Storage Service Authorization";
Authorization: Interface "Storage Service Authorization";
Response: Codeunit "ABS Operation Response";
Containers: Record "ABS Container";
BlobClient: Codeunit "ABS Blob Client";
ContainerContent: Record "ABS Container Content";
ContainerContentText: Text;
StorageAccountName: Text;

begin
    StorageAccountName := 'd365bcblobssd';
    Authorization := StorageServiceAuthorization.CreateSharedKey('YOUR_
STORAGE_ACCESS_KEY');
    ContainerClient.Initialize(StorageAccountName, Authorization);
    //Create a container called mycontainer
    Response := ContainerClient.CreateContainer('mycontainer');
    //List containers
    Response := ContainerClient.ListContainers(Containers);
    if Response.IsSuccessful() then begin
        if Containers.FindSet() then
            repeat
                message('Container Name: %1', Containers.Name);
            until Containers.Next() = 0;
        end
    else
        Message('Error: %1', Response.GetError());

    //Init Blob Client
    BlobClient.Initialize(StorageAccountName, 'mycontainer',
Authorization);
    //Create a blob with text content
    Response := BlobClient.PutBlobBlockBlobText('MyBlob.txt', 'This is the
content of my blob');
    if not Response.IsSuccessful() then
        Message('Blob creation error: %1', Response.GetError());

    //List blobs
    Response := BlobClient.ListBlobs(ContainerContent);
    if Response.IsSuccessful() then begin
        if ContainerContent.FindSet() then
            repeat

```

```

        Message('ContainerContent Name: %1', ContainerContent.
Name);

        BlobClient.GetBlobAsText(ContainerContent.Name,
ContainerContentText);
        Message('%1 content: %2', ContainerContent.Name,
ContainerContentText);
        until ContainerContent.Next() = 0;
    end;
end;
}

```

The above code is quite self-explanatory and uses the Azure Blob Storage AL objects for connecting to the storage account, creating a new container, and appending a new text file to that container.

The result of this code is that a new file is created in our Blob Storage container:

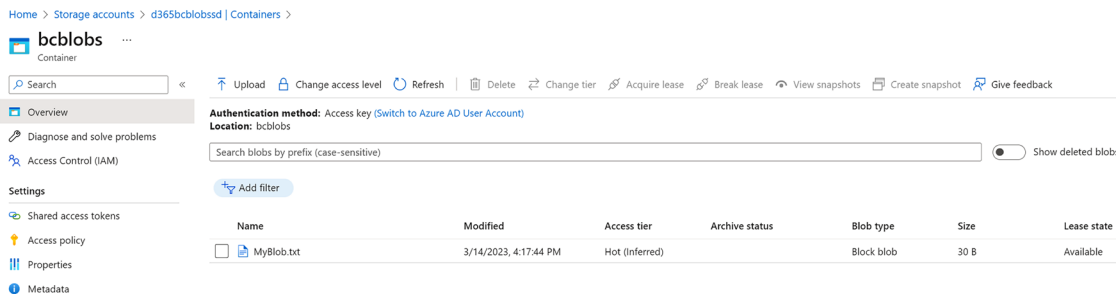


Figure 7.8: File created in Azure Blob Storage

## Handling Dynamics 365 Business Central attachments in Azure Blob Storage

When considering the use of Azure Blob Storage with Dynamics 365 Business Central, a common request is the following: “How can I handle document attachments using Azure Blob Storage? Is it possible to use the standard attachments functionality but save the files in Azure Blob Storage instead of storing them in the database?”

The answer is yes, and to do that you need to work through the following steps:

1. In the `OnBeforeInsertAttachment` event of the `Document Attachment` table, you need to read the incoming file and store it in Azure Blob Storage.
2. You need to delete the `Media` field from the `Document Attachment` table.
3. In the `Document Attachment Details` page you need to add an action to download the attachment file from Azure Blob Storage.
4. When you delete the attachment file, the file should also be deleted from Azure Blob Storage. To handle this you need the `OnBeforeDeleteEvent` of the `Document Attachment` table.

The following codeunit handles the automatic upload of an attachment (loaded via the standard attachment functionality in BC) to Azure Blob Storage, instead of storing it in the database. It performs the insertion of an attachment to Azure Blob Storage and its deletion from the original Media field. The blob is not stored in the Business Central database. No other setup is needed to start using this codeunit:

```
codeunit 50100 "SD Attached Documents Mgt."
{
    //Upload the Blob to Blob Storage
    [EventSubscriber(ObjectType::table, Database::"Document Attachment",
    'OnBeforeInsertAttachment', '', true, true)]
    local procedure OnBeforeImportWithFilter(var DocumentAttachment: Record
    "Document Attachment"; var RecRef: RecordRef)
    var
        ABSBlobClient: codeunit "ABS Blob Client";
        Authorization: Interface "Storage Service Authorization";
        ABSContainerSetup: Record "SD ABS Container Setup";
        StorageServiceAuthorization: Codeunit "Storage Service Authorization";
        InS: InStream;
        OutS: OutStream;
        tempBlob: Codeunit "Temp Blob";
        Filename: Text;
    begin
        ABSContainerSetup.Get;
        Authorization := StorageServiceAuthorization.
        CreateSharedKey(ABSContainerSetup."Shared Access Key");
        ABSBlobClient.Initialize(ABSContainerSetup."Account Name",
        ABSContainerSetup."Container Name", Authorization);
        //Copy from outstream to instream
        tempBlob.CreateOutStream(OutS);
        DocumentAttachment."Document Reference ID".ExportStream(OutS);
        tempBlob.CreateInStream(InS);
        Filename := DocumentAttachment."File Name" + '.' +
        DocumentAttachment."File Extension";
        ABSBlobClient.PutBlobBlockBlobStream(Filename, InS);
    end;

    //When you delete the file, also the file from the Blob Storage must be
    deleted
    [EventSubscriber(ObjectType::table, Database::"Document Attachment",
    'OnBeforeDeleteEvent', '', true, true)]
    local procedure DeleteDocumentAttachment(var Rec: Record "Document
    Attachment"; RunTrigger: Boolean)
    var
```

```

    ABSBlobClient: codeunit "ABS Blob Client";
    Authorization: Interface "Storage Service Authorization";
    ABSContainerSetup: Record "SD ABS Container Setup";
    StorageServiceAuthorization: Codeunit "Storage Service Authorization";
    Filename: Text;
begin
    If RunTrigger then begin
        ABSContainerSetup.Get;
        Authorization := StorageServiceAuthorization.
    CreateSharedKey(ABSContainerSetup."Shared Access Key");
        ABSBlobClient.Initialize(ABSContainerSetup."Account Name",
ABSContainerSetup."Container Name", Authorization);
        Filename := Rec."File Name" + '.' + Rec."File Extension";
        ABSBlobClient.DeleteBlob(Filename);
    end;
end;

//Delete the file from the Media field
[EventSubscriber(ObjectType::table, Database::"Document Attachment",
'OnAfterInsertEvent', '', true, true)]
    local procedure DeleteMediaField(var Rec: Record "Document Attachment";
RunTrigger: Boolean)
    begin
        If RunTrigger then begin
            Clear(Rec."Document Reference ID");
            Rec.Modify();
        end;
    end;
}

```

The following action in the **Document Attachment Details** page handles the download of the attachment from Azure Blob Storage:

```

pageextension 50102 "SD DocumentAttDetailsExt" extends "Document Attachment
Details"
{
    actions
    {
        addlast(processing)
        {

```

```

    action(DownloadBlob)
    {
        ApplicationArea = All;
        Caption = 'Download Blob';
        Image = Download;
        Promoted = true;
        PromotedOnly = true;
        PromotedCategory = Process;
        PromotedIsBig = true;
        Scope = Repeater;
        ToolTip = 'Download the file to your device. Depending on the
file, you will need an app to view or edit the file.';

        trigger OnAction()
        var
            ABSBlobClient: codeunit "ABS Blob Client";
            Authorization: Interface "Storage Service Authorization";
            ABSContainersSetup: Record "SD ABS Container Setup";
            StorageServiceAuthorization: Codeunit "Storage Service
Authorization";

            Filename: Text;
        begin
            ABSContainersSetup.Get;
            Authorization := StorageServiceAuthorization.
CreateSharedKey(ABSContainersSetup."Shared Access Key");
            ABSBlobClient.Initialize(ABSContainersSetup."Account Name",
ABSContainersSetup."Container Name", Authorization);
            Filename := Rec."File Name" + '.' + Rec."File Extension";
            ABSBlobClient.GetBlobAsFile(Filename);
        end;
    }
}
}
}

```

The complete source code for this extension is available from the book's GitHub repository.

When you install this extension, you need to complete the Blob Storage setup on the **SD ABS Container Setup** page (provide the storage account name, container name, and access key).

Then you can, for example, select a Sales Order document and attach a file to it via the standard attachments action:

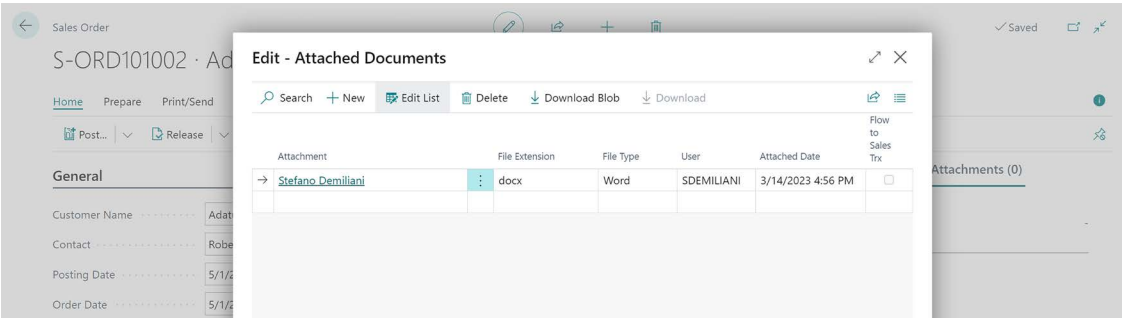


Figure 7.9: Attaching a file to a sales order

When uploaded, the file is automatically saved into the Azure Blob Storage container:

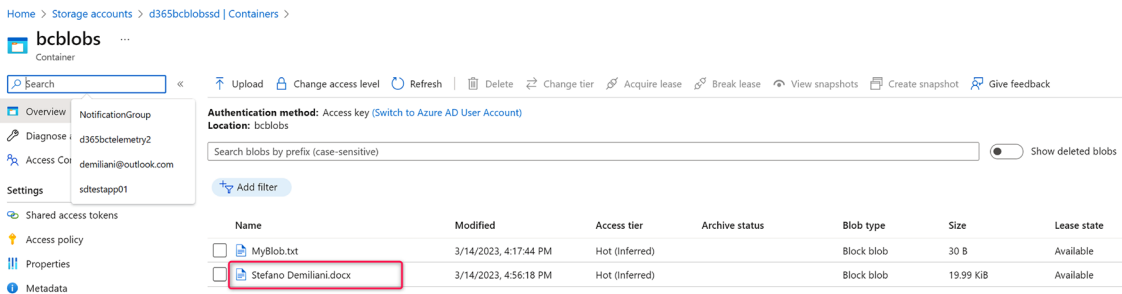


Figure 7.10: Attached file shown in Azure Blob Storage

You can download the attachment directly from the **Document Attachments** page in Dynamics 365 Business Central:

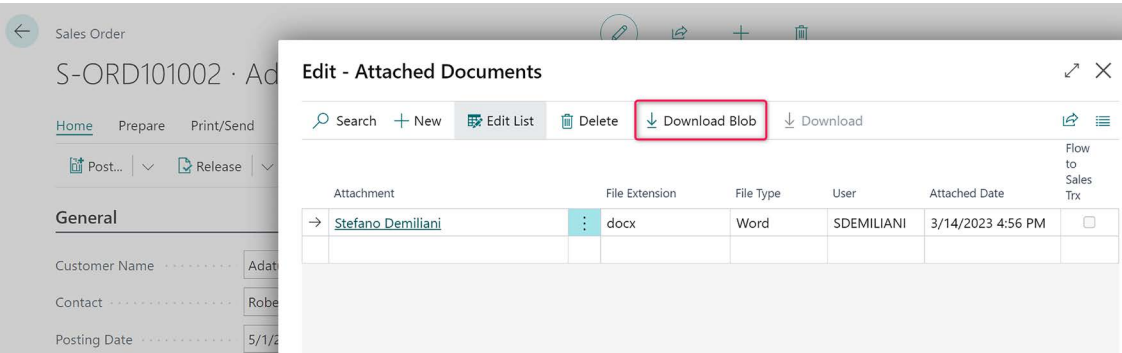


Figure 7.11: Downloading a Blob attachment

And if you delete the file, it's also automatically removed from Azure Blob Storage.

In this way you can handle attachments in every Dynamics 365 Business Central entity directly in Azure Blob Storage in a totally transparent way for the end users.

Azure Blob Storage is also great if you want to save files on the user's corporate network. Dynamics 365 Business Central SaaS cannot directly save a file to a corporate network (you cannot access it from the cloud environment) but you can trigger an Azure Logic Apps workflow that takes the file from Azure Blob Storage and uses the File System connector for on-premises to save the file to a corporate network.

We'll talk about Azure Logic Apps and its connectors in Chapter 14, *Extending Dynamics 365 Business Central with Azure Services*.

## Using Azure file shares from AL

Starting from Dynamics 365 Business Central version 23.2, a new native **Azure file share AL module** is available. This module was created with the same pattern as the **Azure Blob Storage AL module** and permits the reading/writing of files to an Azure file share instance.

**Azure Files** and **Azure Blob Storage** both offer ways to store large amounts of data in the cloud, but each is most appropriate for slightly different use cases.

**Azure Blob Storage** is useful for massive-scale, cloud-native applications that need to store unstructured data. To maximize performance and scale, Azure Blob Storage is a simpler storage abstraction than a true file system. You can access Azure Blob Storage only through REST-based client libraries (or directly through the REST-based protocol).

**Azure Files** is specifically a file system in the cloud. Azure Files has all the file abstracts that you know and love from years of working with on-premises operating systems. Like Azure Blob Storage, Azure Files offers a REST interface and REST-based client libraries. Unlike Azure Blob Storage, Azure Files offers SMB access to Azure file shares. By using SMB, you can mount an Azure file share directly on Windows, Linux, or macOS, either on-premises or in cloud VMs, without writing any code or attaching any special drives to the file system. You also can cache Azure file shares on on-premises file servers by using Azure File Sync for quick access, close to where the data is used.

The module uses the common **Storage Service Authorization** interface for authentication with the storage account. For authentication, you can use a shared key or a SAS token.

Then it adds some operational codeunits including the **AFS File Client**, **AFS Operation Response**, and **AFS Optional Parameters** codeunits, and a table called **AFS Directory Content** for handling the files in the storage.

Here is an example of a page action for writing files to an Azure file share:

```
action(SendFile)
{
    ApplicationArea = All;
    Caption = 'Write File to File Share';
    Image = SendAsPDF;
    Tooltip = 'Writes a file to the Azure File Share';

    trigger OnAction()
    var
```

```

StorageServiceAuthorization: Codeunit "Storage Service
Authorization";
AzureFileShareHandler: Codeunit "SD Azure File Share
Handler";
FileStream: InStream;
FileName, FilePath : Text;
begin
    if not UploadIntoStream('', '', '', FileName, FileStream)
then
    exit;
    AzureFileShareHandler.WriteFileToShare(StorageAccount,
    FileShare, StorageServiceAuthorization.UseReadySAS(SASToken), FilePath,
    FileStream);
    Message('File %1 saved succesfully.', FileName);
end;
}

```

And here is the implementation of the WriteFileToShare method (that saves the file to the Azure file share container):

```

procedure WriteFileToShare(StorageAccount: Text; FileShare: Text;
Authorization: Interface "Storage Service Authorization"; FilePath: Text; var
FileContent: InStream)
var
    AFSFileClient: Codeunit "AFS File Client";
    AFSOperationResponse: Codeunit "AFS Operation Response";
begin
    AFSFileClient.Initialize(StorageAccount, FileShare, Authorization);
    AFSOperationResponse := AFSFileClient.CreateFile(FilePath,
FileContent);
    if not AFSOperationResponse.IsSuccessful() then
        Error(AFSOperationResponse.GetError());

    AFSOperationResponse := AFSFileClient.PutFileStream(FilePath,
FileContent);
    if not AFSOperationResponse.IsSuccessful() then
        Error(AFSOperationResponse.GetError());
end;

```

In the same way, you can read files from the Azure file share. Here is a page action for reading a file:

```

action(GetFileAsStream)
{
    ApplicationArea = All;
    Caption = 'Get File from Share';
    ToolTip = 'Gets a file as a stream from Azure File Share';
    Image = Download;

    trigger OnAction()
    var
        StorageServiceAuthorization: Codeunit "Storage Service
Authorization";
        AzureFileShareHandler: Codeunit "SD Azure File Share
Handler";
        FilePath: Text;
    begin
        AzureFileShareHandler.GetFileFromShare(StorageAccount,
        FileShare, StorageServiceAuthorization.UseReadySAS(SASToken), FilePath);
    end;
}

```

And here is the implementation of the GetFileFromShare method (that reads a file from the file share):

```

procedure GetFileFromShare(StorageAccount: Text; FileShare: Text;
Authorization: Interface "Storage Service Authorization"; FilePath: Text)
var
    AFSFileClient: Codeunit "AFS File Client";
    AFSOperationResponse: Codeunit "AFS Operation Response";
    FileContents: InStream;
begin
    AFSFileClient.Initialize(StorageAccount, FileShare, Authorization);
    AFSOperationResponse := AFSFileClient.GetFileAsStream(FilePath,
    FileContents);
    if not AFSOperationResponse.IsSuccessful() then
        Error(AFSOperationResponse.GetError());

    DownloadFromStream(FileContents, '', '', '', FilePath);
end;

```

With Azure file shares you can map a file share in the cloud as a local drive on the client machine. This allows you to save a file from Dynamics 365 Business Central to the file share and the file will automatically be visible on the client machine. The same goes for the opposite direction (a file saved locally on the mapped share drive is automatically uploaded to the cloud file share).



Figure 7.12: Azure file share shown on client machine

Here you can find a Powershell script that maps an Azure file share instance as a local drive: <https://github.com/demiliani/PowershellCloudScripts>.

## Summary

In this chapter, we saw how to handle files and unstructured information from Dynamics 365 Business Central code. We also encountered practical examples of how to use streams, blobs, XMLports, XML and JSON files, along with Isolated Storage for handling sensitive information, and Azure Blob Storage and Azure file shares for moving all unstructured objects outside the ERP database (in order to reduce usage of the tenant's storage capacity).

Saving unstructured files outside of the database is a best practice that you should start adopting because it gives you a lot of possibilities (not only for saving storage but also creating document workflows, sharing documents with external applications, and so on).

By using the Azure file share module you can give your users a way of managing files very similar to that in a traditional on-premises application. The user can simply save a file in a local folder, without knowing that the file is stored in the Azure cloud.

In the next section, we'll see how to create new reports with AL code and how to extend existing Dynamics 365 Business Central reports.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>





# 8

## Report Development

In this book, we have been introduced to and analyzed a vast variety of AL language objects, and we have seen how to develop simple and complex extensions with them.

In this chapter, we will deep dive into a couple of specific objects and look at their properties and how to use them proficiently. These objects are the **report** and **report extension** objects.

An overview, with pros and cons, will be provided regarding which tool to use to design and develop datasets and layouts, such as Microsoft Word and Excel layouts, or Microsoft Report Builder for **Report Definition Language (RDL)** layouts. We will discuss the main shortcuts, tips, and tricks related to dataset development with Visual Studio Code and all report layouts.

In this chapter, we aspire to provide you with the confidence you need to develop Dynamics 365 Business Central report and report extension objects, explain when to use one object or the other and how to make reports perform to their best, and help you troubleshoot the most common issues in this area.

In this chapter, we will cover the following topics:

- Anatomy of the AL report and report extension objects
- Tools to use for RDL, Word, and Excel layouts
- Feature limitations when developing an RDL or Word layout report
- Understanding report performance considerations

### Anatomy of the report object

Requests for new reports come from every department in client organizations and in many different forms. Most of the time, users have an idea in mind of how they would like the data to be shown.

Nevertheless, a report developer should always keep in mind some important points.

All the important points to keep in mind are related to data:

- **Retrieval:** A good report developer should have a good understanding of the business process (how data is created, modified, and deleted) and data topology (where data is stored). Data can be retrieved from heterogeneous resources that cannot be directly stored in Dynamics 365 Business Central tables. As an example, you might want to run an HTTP call to a web service to gather some data outside the database and store it in a physical or temporary table before processing it.
- **Processing:** Some of the data presented could be the result of data aggregation, calculated from different fields, or even from the concatenation of values from different tables. The result of data retrieval and processing generates the dataset.
- **Presentation:** Datasets are sent from the application to the Report Viewer component, which takes care of data rendering and presentation. Together with the dataset, a report definition file (`Report.rdlc`) is sent to the Report Viewer component to build up the content of the report. The report definition file contains the metadata structure and rules to render the report. Despite its extension (`.rdl`), this is simply an XML-formatted flat file. Tools such as the Report Builder component or Visual Studio can digest the XML file and create a presentation of the report structure in a more human-readable way. Every action that is taken in this designer has the consequence of editing and changing the `report.rdl` XML file.

With Dynamics 365 Business Central, it is also possible to design reports to perform only data retrieval and processing, typically committing changes in tables (insert, modify, or delete) as a final result of the process. No data is typically presented to the users; hence no layout is needed, and therefore no dataset will be created.

Reports can then be grouped into two main categories: processing-only and dataset-based.

Reports that are processing-only do not have any layout. They also typically do not have any columns defined in the dataset and are only used to process data. Quite often, the same result could be achieved using codeunit objects instead, since these are very simple code repositories and do not have any graphical or user interface (UI) interaction. To give you a simple example, you could create a processing-only report with a data item that loops all the customer records and prints a flat JSON file with the customer number, name, and email. The same could be achieved by implementing a codeunit with a function that loops through a set of customers. Within this loop, it is possible to write a flat JSON file with the same information.

The pros and cons of using a processing-only report versus a codeunit have been tabulated here:

	Processing-Only Report	Codeunit
UI Interaction	It has a pre-built artifact to collect filters and specify processing settings.	It does not have a pre-built structure.
Easy to Implement	It is quick to implement. The data item looping construct is predefined.	There is more development activity required in building a loop.

Flexibility	It is limited to data item triggers.	It is more flexible. It does not stick to a simple and fixed iterating structure, but rather is open and offers numerous development possibilities.
Performance	It has worse performance at runtime.	Codeunits have the potential for better performance, since you control the processing.

Table 8.1: Process-only report comparison table

A report object has the following tree structure:

```
|---Properties
|---Dataset
| |---Data items:
| | |---Properties
| | |---Columns
| | | |---Properties
| | | |---Triggers
| | |---Triggers
|---Request Page
|Layout
| | |---Columns
| | | |---Group
| | | | |---Field
| |---Actions
|---Labels
|---Triggers
|---Rendering
| |---Layout
| | |---Properties
```

After installing the standard **AL Language extension** in Visual Studio Code, we can use the `treport` snippet to create a skeleton of a report and inspect all the different items related to the main content areas: the dataset and the request page. Layouts are just references to the corresponding output within the report objects (RDL, Word, or Excel files, typically), and these are created with tools other than Visual Studio Code. We will work with them later in this chapter.

## The report extension object

The report extension object has been introduced in AL with the main purpose of avoiding the cloning of an entire report and just to add one or a few fields to an existing dataset.

Just to give you an idea, within the standard system and base application, there are no report extension objects, but if you have some of your customers asking you to add just a simple static field in a document or something similar, then the report extension object could be the right development choice.

The main advantage of the report extension is that you can demand the maintenance of the report logic by the party that sources the report (typically Microsoft), while isolating minor changes in an equivalent small object.

The main drawback of this object is that when it comes to editing, changing, or printing out values that are not directly bound to a table (e.g., global variables and calculated values within the report itself), then you should choose to clone or design the entire report object from scratch.

To summarize, the report extension object can be used to:

- Add columns to existing data items.
- Add new simple data items (with not that much code in them).
- Add simple code to triggers.
- Add columns to the request page.
- Add report layouts.
- Add custom telemetry signals to existing reports.

After exploring the anatomy of report and report extension objects, it is time to see the tools that are used to create and edit RDL, Word, and Excel layouts.

## Tools to use for RDL, Word, and Excel layouts

Visual Studio Code does not have a valid extension—yet—that would replace the best-in-class RDL report editor, which is supported by the Dynamics 365 Business Central development team. With a release every six months, the application is always up to date and, at the time of writing, it deploys Report Viewer 15 (from SQL Server 2022 Reporting Services) and the latest RDL 2019 schema-based syntax.

To develop an RDL layout report, you have two choices:

- **Report Builder for SQL Server 2019:**  
<https://www.microsoft.com/en-us/download/details.aspx?id=53613>
- **Visual Studio 2019 with the Microsoft RDLC Report Designer for the Visual Studio extension installed:** <https://go.microsoft.com/fwlink/?linkid=857038>



Note that you need to have Visual Studio 2019 installed first. This is a plugin for Visual Studio; it is not a stand-alone report designer.

To find out more, please visit the following official reference and this useful blog:

- <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/deployment/system-requirements-business-central-v21>
- <https://thinkaboutit.be/2022/02/how-do-i-find-the-correct-version-of-visual-studio-to-design-business-central-rdlc-report-layouts/>

Within this chapter, we will use Report Builder, but feel free to also use Visual Studio, if you prefer.

The Word layout feature is built around the latest **Aspose.Words** (<https://products.aspose.com/words>) component and is implemented in the application by the server runtime team. Designing and editing must be done with a version of Microsoft Word that supports XML mappings. Minimum system requirements specify that we should use **Microsoft Word 2019** or later.

As for Excel layouts, as per current system requirements, these should be created and edited using **Microsoft Excel 2019** or later.

## The RDL, Word, and Excel layout features

To explain and show some of the most important RDL, Word, and Excel layout features are supported by Dynamics 365 Business Central, we will go through a step-by-step example.

In *Chapter 4, Developing a Customized Solution for Dynamics 365 Business Central*, we extended the Item Ledger Entry table by creating the Customer Category field for statistics, and in the subsequent chapter, we finalized the code for the extensibility of our demo extension. You can download the extension from the book's repo: <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition/tree/main/CH05/PacktDemoExtension>

It is now time to create a report in this demo app that uses this extended field for sales analysis purposes. Create a new directory in your extension called `. \Src\CustomerCategory\report`. Let us learn how to do this.

## Part 1 – Designing the dataset

The dataset is designed within the report by specifying the data item field's columns and their properties. They have considerable influence from a qualitative perspective (in which format is data exported?) and from a quantitative perspective (how many rows are processed?). Both have a clear impact on report performance.

A short example is represented by decimal data types. When specifying a decimal data type in a dataset, two fields are always included: the decimal data and its formatting. This means that, together with the decimal field, you always have a repeated text variable tightly bound to it. This would increase the dimension of the dataset and, consequently, influence report performance.

What could be the definition of a Dynamics 365 Business Central dataset? Here is mine:

“A dataset is like a table in the client memory whose columns are made of all of the column fields defined in the dataset section and whose rows are all valid (not skipped) records that are processed in the data items.”

Let us design/create our report dataset:

1. Create a new file in the .\Src\CustomerCategory\report folder called ItemLedgerEntryAnalysis.report.al.
2. Type treport to enable the report snippet.
3. Add the following properties, the data item, and its columns:

```
report 50111 "Item Ledger Entry Analysis"
{
    DefaultRenderingLayout = LayoutRDL;

    Caption = 'Item Ledger Entry Analysis';
    UsageCategory = ReportsAndAnalysis;
    ApplicationArea = All;

    dataset
    {
        dataitem("Item Ledger Entry"; "Item Ledger Entry")
        {
            column(ItemNo_ItemLedgerEntry; "Item Ledger Entry"."Item
No.")
            {
                IncludeCaption = true;
            }
            column(PostingDate_ItemLedgerEntry; "Item Ledger
Entry"."Posting Date")
            {
                IncludeCaption = true;
            }
            column(EntryType_ItemLedgerEntry; "Item Ledger Entry"."Entry
Type")
            {
                IncludeCaption = true;
            }
            column(CustCatPKT_ItemLedgerEntry; "Item Ledger Entry"."PKT
Customer Category Code")
            {
                IncludeCaption = true;
            }
            column(DocumentNo_ItemLedgerEntry; "Item Ledger
Entry"."Document No.")
            {
```

```

        IncludeCaption = true;
    }
    column(Description_ItemLedgerEntry; "Item Ledger Entry".
Description)
    {
        IncludeCaption = true;
    }
    column(LocationCode_ItemLedgerEntry; "Item Ledger
Entry"."Location Code")
    {
        IncludeCaption = true;
    }
    column(Quantity_ItemLedgerEntry; "Item Ledger Entry".
Quantity)
    {
        IncludeCaption = true;
    }
    column(COMPANYNAME; CompanyName)
    {
    }
    column(includeLogo; includeLogo)
    {
    }
    column(CompanyInfo_Picture; CompanyInfo.Picture)
    {
    }
    }
}

```

Remember that you can download the extension from the book's repo: <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition/tree/main/CH05/PacktDemoExtension>

After the data item section, we add the rendering section, some labels, and variables:

```

rendering
{
    layout(LayoutRDL)
    {
        Type = RDLC;
        Caption = 'LayoutRDL';
        Summary = 'Item Ledger Entry Analysis RDL Report';
        LayoutFile = '.\Src\CustomerCategory\report\
ItemLedgerEntryAnalysis.report.rdl';
    }
}

```

```

    }
    layout(LayoutWord)
    {
        Type = Word;
        Caption = 'LayoutWord';
        Summary = 'Item Ledger Entry Analysis Word Report';
        LayoutFile = '.\Src\CustomerCategory\report\
ItemLedgerEntryAnalysis.report.docx';
    }

    layout(LayoutExcel)
    {
        Type = Excel;
        Caption = 'LayoutExcel';
        Summary = 'Item Ledger Entry Analysis Excel Report';
        LayoutFile = '.\Src\CustomerCategory\report\
ItemLedgerEntryAnalysis.report.xlsx';
    }
}

labels
{
    PageNo = 'Page';
    BCReportName = 'Item Ledger Entry Analysis';
}

var
    CompanyInfo: Record "Company Information";
    includeLogo: Boolean;

```

4. Build (*Ctrl + Shift + B*) the extension. This will automatically generate three empty report layout files (with extensions *.rdl*, *.docx*, and *.xlsx*) in the *.\Src\CustomerCategory\report* folder. (If you already have the repo, these files already exist.)
5. In Visual Studio Code, right-click the RDL file and choose the option **Open externally**. It will open in whichever program you selected to work with the *.rdl* file extension by default. In this example, we will work with **Microsoft SQL Server Report Builder 2019**.
6. If it is not already enabled, be sure to have the **Report Data** option checked in the **View | Show/Hide** ribbon menu in your Report Builder 2019 instance. In the **Report Data** pane, expand Parameters and Datasets.

You will notice that Parameters items are labels and field captions (specified by the `IncludeCaption=true` property in the dataset). `DataSet_Result` shows the entire dataset's definition transposed into the Report Builder IDE. Here is a screenshot of the **Report Data** pane:

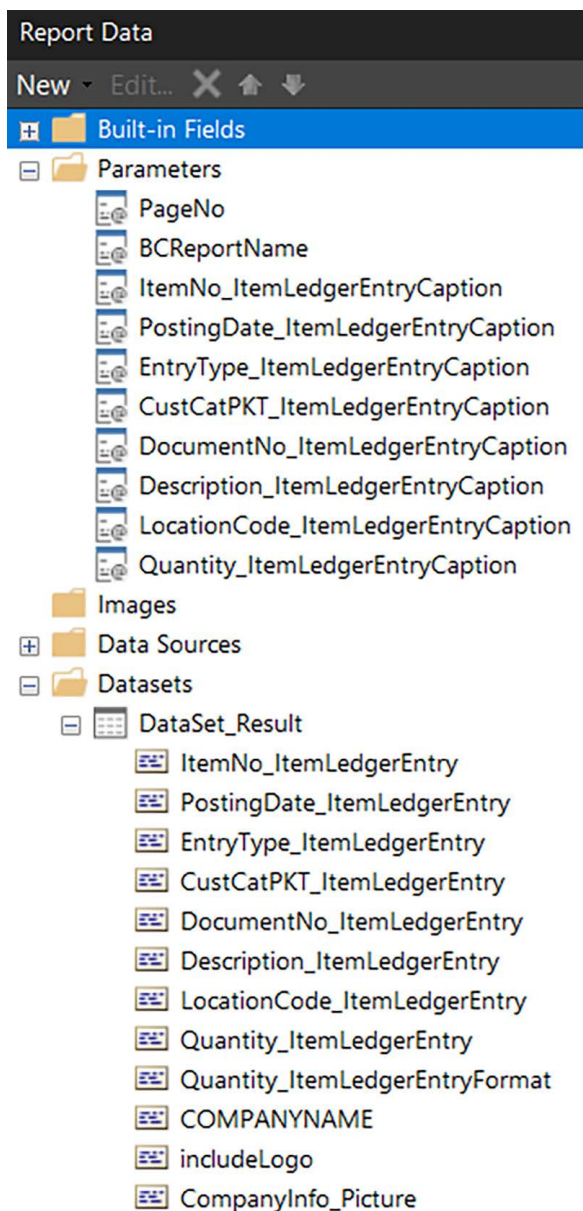


Figure 8.1: Report Data pane

## Part 2 – Creating a simple RDL layout

RDL layout development is one of the several layout design options you have for Dynamics 365 Business Central. For most cases, the simple report builder is sufficient. But, compared to the report builder, Visual Studio has more capabilities. Report layout design with Visual Studio 2019 (plus the Microsoft RDLC Report Designer for Visual Studio extension) has a fully fledged development experience compared to Report Builder. For example, it has a **Document Outline** window, which shows a hierarchical view of the controls in the layout and lets you jump from one control to another quickly.

Specifically targeted at RDL layout development, you can find quite exhaustive official documentation from SQL Server Reporting Services, official courseware, or third-party books. This section contains some particularly good development references if you would like to completely master the RDL layout for Dynamics 365 Business Central. Even if they mostly come from earlier versions of Dynamics NAV or SQL Server Reporting Services, they still contain great hints, and you should spare them a spot in your personal library.

Here are some RDL layout development references. While they are outdated nowadays, the functionality they describe is fundamentally the same:

Microsoft Dynamics NAV 2015 Professional Reporting	Steven Renders, Packt Publishing
Microsoft Dynamics NAV 2009: Professional Reporting	Steven Renders, Packt Publishing
Microsoft Dynamics NAV 2009 INSIDE Reporting	Rene Gayer, MBST-Training*
Professional Microsoft SQL Server 2008 Reporting Services	Stacia Misner, Microsoft Press US

Table 8.2: Further reading on RDL layout development

\*Please find this resource here: <https://portal.learn4d365.com/unit/view/id:10733>

### Part 2.1 – Creating the RDL report header

In this section, we will create the report header step by step:

1. In **Report Builder**, let us set up the report properties: right-click anywhere in the gray development area and select **Report Properties**:

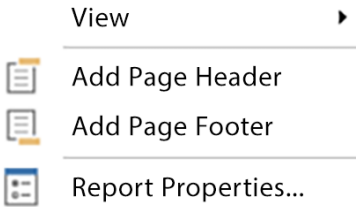


Figure 8.2: Report Properties option

2. Change the page setup parameters, as follows:

Paper size

Orientation:

A

A

PortraitLandscape

Paper size:Width:Height:

Letter8,5in11in

Margins

Left:Right:

0,69444in0,41667in

Top:Bottom:

0,41667in0,58333in

Figure 8.3: Page setup parameters

- 3. Let us add a page header: right-click anywhere in the gray development area and select **Add Page Header**.
- 4. Right-click anywhere in the page header and select **Header Properties**.
- 5. Edit **Page Header Properties** as follows:

Page Header Properties

GeneralFillBorder

Change header properties for the page

☒ Display header for this page

Print options:

☒ Print on first page☒ Print on last page

Height:

0,63542in

Figure 8.4: Page Header Properties

- 6. Click on the body section and change the body size property as follows:
  - **Width: 7.21205in**
  - **Height: 1.93403in**
- 7. Now, let us directly add some report item controls to the header. Right-click anywhere in the report header and select **Insert | Textbox** in the page header. Perform this action six times.
- 8. Change the following properties for the textboxes:

Name	Value	Size	Location	TextAlign
txtReportName	=Parameters!BCReportName .Value	7.5 cm; 0.423 cm	0 cm; 0.0005 cm	Default
txtCompanyName	=Fields!COMPANYNAME .Value	7.5 cm; 0.423 cm	0 cm; 0.45878 cm	Default
txtExecutionTime	=Globals!ExecutionTime	3.15 cm; 0.423 cm	15 cm; 0.0005 cm	Right
txtPageNoLabel	=Parameters!PageNo.Value	1.25271 cm; 0.423 cm	16.44729 cm; 0.4235 cm	Left
txtPageNumber	=Globals!PageNumber	0.45 cm; 0.423 cm	17.7 cm; 0.45878 cm	Right
txtUserID	=User!UserID	3.15 cm; 0.423 cm	14.8868 cm; 0.91298 cm	Right

Table 8.3: Textbox properties

For each textbox, set **VerticalAlign** to **Middle**, **CanGrow** to **False**, and **Font** to **Arial**.  
Set the font size to 8pt and **Bold** for txtReportName, and 7pt and **Default** for everything else.  
This is what it should look like in the report header section:



To add an item to the report: on the Insert tab, click the item and drag dataset fields to the item.

Figure 8.5: Report header layout

Part 2.2 – Adding a table control to the RDL report body

In this section, we will add a table control in the report body to display data in a tabular format:

1. Right-click somewhere in the middle of the report body and choose **Insert**. Select the **Table** control from the menu. Keep the table small since we will need to add some extra columns and resize the width manually.
2. Select the last table column, right-click and select **Insert Column**, and choose **Right**:

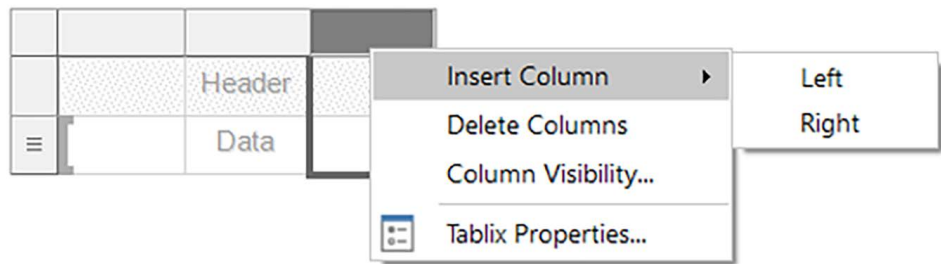


Figure 8.6: Options for Insert Column

3. Repeat *step 2* three more times so that we have seven columns in total.
4. Keep the table aligned by changing the following table properties:
  - **Location:** 0.02584 in; 0.18403in
  - **Size:** 7.06693 in; 0.48958in
5. Change the column width property for each column, from left to right, as follows:

Column	1	2	3	4	5	6	7
Width (cm)	1,905	2,222	2,593	2,990	3,373	2,620	1,798

Table 8.4: Column width values

This should be the current layout result:

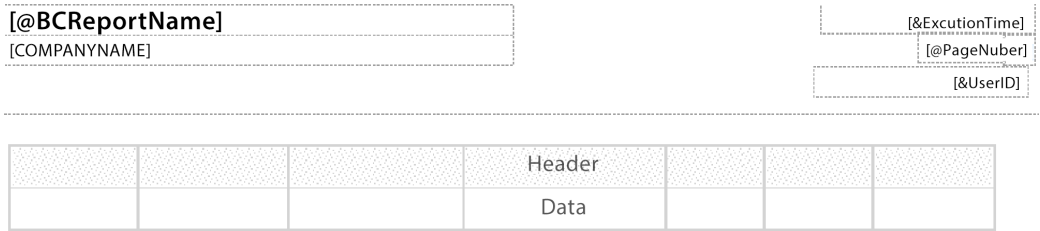


Figure 8.7: Column layout

6. It is now time to set the table properties appropriately and bind them to the Dynamics 365 Business Central dataset. Select the table (once you have clicked on the table, a gray column/row area appears, and in a Microsoft Excel-like style, just click in the top left of this area to select the whole table).

7. In the **Properties** window, click the **Property Pages** button.
8. Change the values in the **General** tab as follows and click **OK**:
  - **Name:** tableItemLedgerEntry
  - **Dataset name:** Dataset\_Result

Now, the table is bound to the appropriate dataset, and it has a self-explanatory name. Giving an appropriate name to every single control in the RDL layout is useful because you can find out the purpose of the control and where it is located immediately. Now, let us bind every table control to the dataset caption and field values.

9. For every single text box in the first row of the table (the table header row), open the **Property Pages** window and change the **Value** property of the first seven table header textboxes, as follows:

Name	Value
txtItemNoCap	=Parameters!ItemNo_ItemLedgerEntryCaption.Value
txtPostingDateCap	=Parameters!PostingDate_ItemLedgerEntryCaption.Value
txtCustCatPKTCap	=Parameters!CustCatPKT_ItemLedgerEntryCaption.Value
txtDocumentNoCap	=Parameters!DocumentNo_ItemLedgerEntryCaption.Value
txtDescriptionCap	=Parameters!Description_ItemLedgerEntryCaption.Value
txtLocationCodeCap	=Parameters!LocationCode_ItemLedgerEntryCaption.Value
txtQuantityCap	=Parameters!Quantity_ItemLedgerEntryCaption.Value

*Table 8.5: Table header values*

10. Let us bind table body text box controls to the dataset fields. For every text box in the second row of the table (the table body), open the **Property Pages** window and change the following properties:

Name	Value	Format
txtItemNo	=Fields!ItemNo_ItemLedgerEntry. Value	
txtPostingDate	=Fields!PostingDate_ ItemLedgerEntry.Value	
txtCustCatPKT	=Fields!CustCatPKT_ ItemLedgerEntry.Value	
txtDocumentNo	=Fields!DocumentNo_ ItemLedgerEntry.Value	
txtDescription	=Fields!Description_ ItemLedgerEntry.Value	
txtLocationCode	=Fields!LocationCode_ ItemLedgerEntry.Value	
txtQuantity	=Fields!Quantity_ ItemLedgerEntry.Value	=Fields!Quantity _ItemLedgerEntryFormat. Value

Table 8.6: Property Pages properties

11. Create an alternate line color to make it easier to read. In the table details, select the seven textboxes mentioned in the previous table, regarding the data section, and add the following values to the following properties in each of these text boxes:

```
Background Color: =iif(LineNumber(Nothing) mod 2, "AliceBlue", "White")
TextAlign: Right
```

12. Conditionally format the **Color** property for the txtQuantity text box. Select txtQuantity and change the **Color** property as follows:

```
Color: =iif(Fields!Quantity_ItemLedgerEntry.Value <= 0, "Red", "Black")
```

13. Enable the ability to display the table header at the beginning of every page to improve report readability. Click on the small down arrow in **Column Groups** and enable **Advanced Mode**.



Figure 8.8: Advanced Mode

14. Select the **(Static)** group in **Row Groups**:



Figure 8.9: Row Groups options

Change the properties for the **(Static)** group as follows:

- **KeepTogether:** True
- **RepeatOnNewPage:** True

## Part 3 – Understanding grouping

By the term **grouping**, we mean the capacity of aggregate result sets based on one or more discriminant elements. Grouping is typically used to show totals per group and/or aggregate and calculate totals (typically using sum formulas). The grouping feature is used typically within controls that implement the data region scope, such as table, matrix, list, or chart.

In this section, we will create group totals for the table control of our report:

1. In the **Row Groups** section, right-click on the **(Details)** static row group and select **Add Group**. Choose **Parent Group....**
2. A pop-up window should appear, asking you to provide a grouping element. Select the **[Item-No\_ItemLedgerEntry]** field value and choose to add both a group header and a group footer. Then, click **OK**:

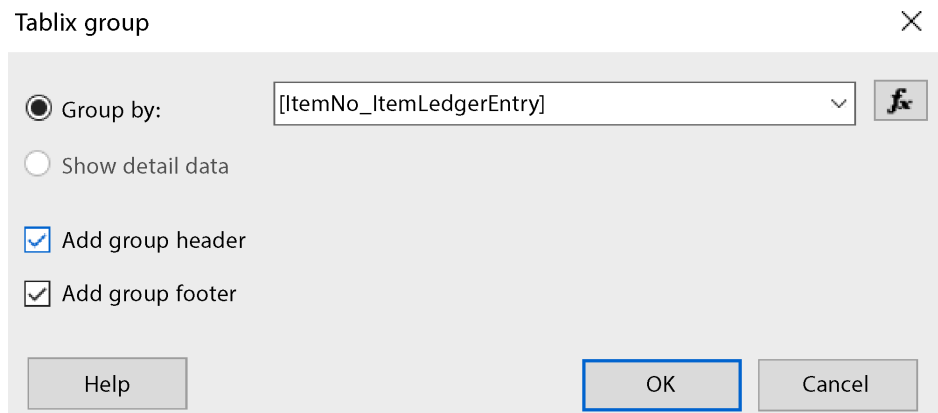


Figure 8.10: Grouping element pop-up

3. Select the group we just created (typically named `ItemNo_ItemLedgerEntry` by default), right-click, and select **Group Properties....**
4. In the **Group Properties** window, change the name to `ItemNoGroup` and click **OK**.
5. Select `ItemNoGroup`, right-click, and select **Add Group** and **Child Group....**
6. A pop-up window appears, asking you to provide a grouping element. Select **CustCatPKT\_ItemLedgerEntry**, choose to add just the group header (no group footer), and click **OK**.
7. Select the group we have just created, right-click, and select **Group Properties....**
8. In the **Group Properties** window, change **Name** to `CustCatPKTNoGroup`. Then, go to the **Advanced** tab and set `CustCatPKT_ItemLedgerEntry` in the **Recursive Parent** box. Click **OK** to confirm this:

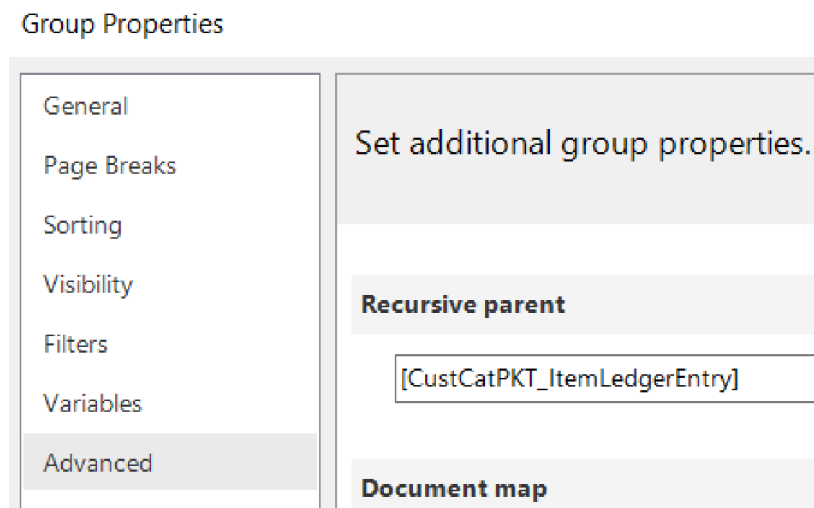


Figure 8.11: Advanced group properties

9. All these operations should have created, by default, two extra unwanted columns in the table to display grouping elements. Select the first two columns and then right-click and choose **Delete Columns**.
10. The automatic action of adding two extra columns should have automatically enlarged the **report body**: bring this back to its original size. Set the following value:

Size: 7.21205in; 1.93403in

11. Let us add group caption labels. Select the first cell in the **ItemNoGroup** header row, as shown in the following screenshot:

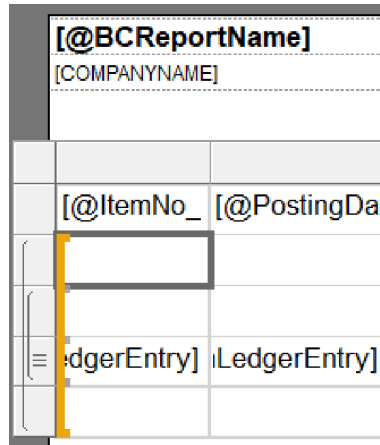


Figure 8.12: ItemNoGroup column cell

12. Change its properties as follows:

**Name:** txtItemNoGroup  
**Value:** =Fields!ItemNo\_ItemLedgerEntry.Value  
**BackgroundColor:** LightBlue

13. Select the third cell in the header column and change its properties as follows:

**Name:** txtCustCatPKTGroup  
**Value:** =Fields!CustCatPKT\_ItemLedgerEntry.Value  
**BackgroundColor:** LightSteelBlue

14. Select the last cell the bottom right, in the ItemNoGroup footer row, and change its properties as follows:

**Name:** txtSumQuantity  
**Value:** =Sum(Fields!Quantity\_ItemLedgerEntry.Value)  
**Color:** =iif(Sum(Fields!Quantity\_ItemLedgerEntry.Value) <= 0, "Red", "Black")  
**Font:** Arial; 10pt; Default; Bold; Default  
**Format:** =Fields!Quantity\_ItemLedgerEntryFormat.Value

15. Now, for the rest of the cells in all the grouping rows (both headers and footers), remove the `BackgroundColor` formula, `=iif(LineNumber(Nothing) mod 2, "AliceBlue", "White")`, that has been automatically copied from the details group. When the formula is deleted from the **Properties** window, **BackgroundColor** will default to **No Color**.
16. We are almost finished with groupings. Let us add the last touch to the totals. Select the sixth cell in the **ItemNoGroup** footer row, click on the **Property Pages** pop-up window for this text box, and change the **Border** properties, as shown in the following screenshot:

Text Box Properties

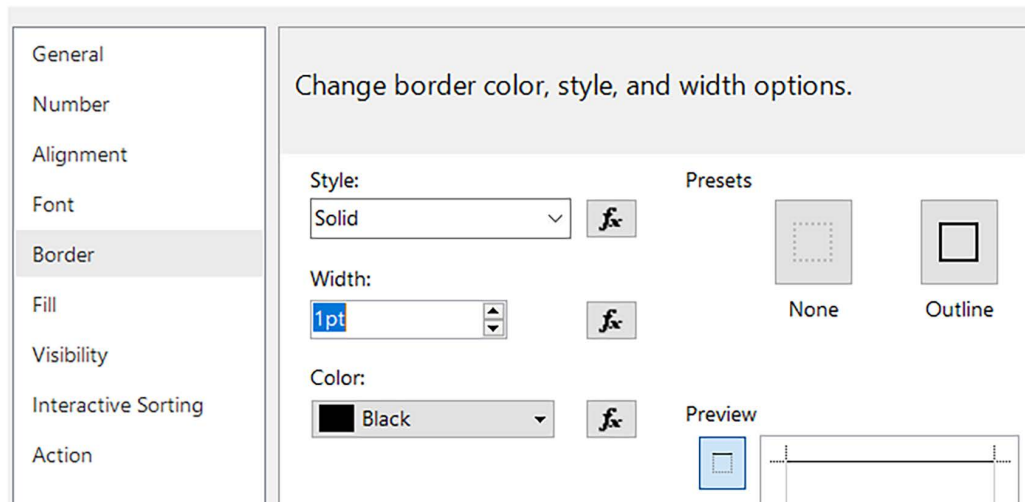


Figure 8.13: Cell border properties

17. Repeat *step 15* for the seventh cell (`txtSumQuantity`).
- Enable the ability to display the table header if there is more than one page to be rendered. Click on the small down arrow in **Column Groups** and enable **Advanced Mode**. Select the **(Static)** group in **Row Groups** and change the properties of the Static member, as follows:

```
KeepTogether: True
KeepWithGroup: After
RepeatOnNewPage: True
```

18. Finally, save the layout.

## Part 4 – Building a simple request page

If report objects have been set with `UseRequestPage = true`; (the default value), then a request page will be shown to the user so that they can set filters, gather user information, and populate AL variables or parameters that influence the processing and output of the report.

Within a request page, you can also add actions to perform some extra activities before running the report. Typical examples include a shortcut to run a page to check for some specific setup or an action that performs pre-processing tasks before setting request page variables.

In Visual Studio Code, add (or change, if you have used the `treport` snippet) the `requestpage` section after the `Dataset` section as follows:

```
requestpage
{
    layout
    {
        area(content)
        {
            group(Options)
            {
                Caption = 'Options';
                field(includeLogo; includeLogo)
                {
                    Caption = 'Include company logo';
                    ApplicationArea = All;
                }
            }
        }
    }
}
```

Also add the `OnPreReport()` trigger section after the `Rendering` section. Consider the following code to get the company information record and from this record compute the company logo:

```
trigger OnPreReport()
begin
    if includeLogo then begin
        CompanyInfo.Get();
        CompanyInfo.CalcFields(Picture);
    end;
end;
```

## Part 5 – Adding database images

In this section, we will add the ability to display images at runtime on our report:

1. Go back to **Report Builder** and the RDL layout. In the center of the report header, right-click and select **Insert | Image**.

2. The **Image Properties** pop-up window will load. In the **General** tab, change **Name** to `imgCompanyLogo` and input the parameters shown in the following screenshot:

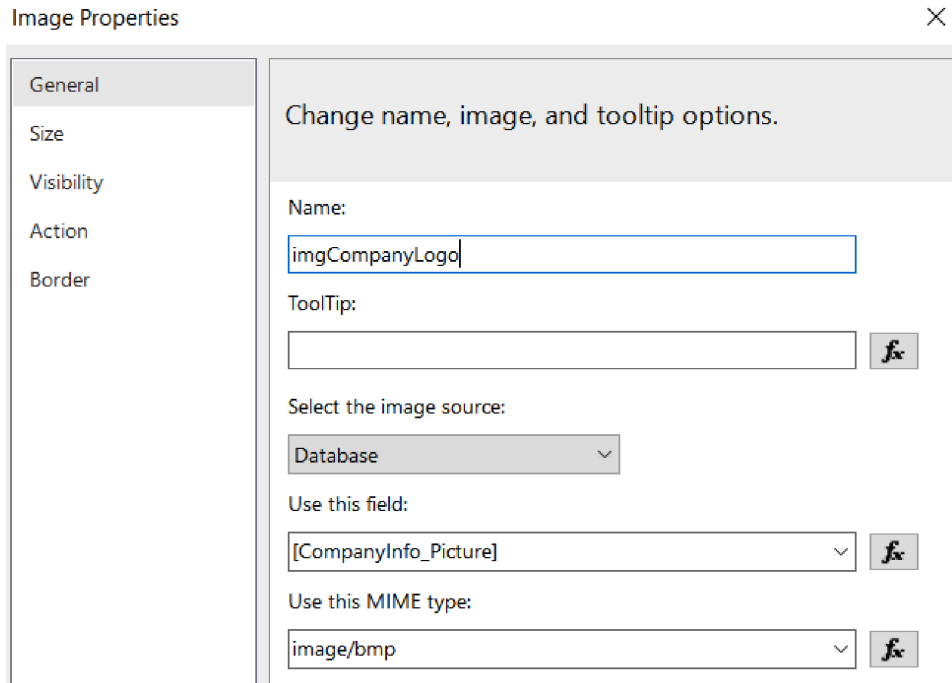


Figure 8.14: Image Properties pop-up

3. In the **Visibility** tab, change the **Display** option to **Show or Hide based on an expression** and add the following formula:


```
=iif(Fields!includeLogo.Value,iif(Len(Convert.
ToString(Fields!CompanyInfo_Picture.Value))>0,False,True),True)
```

This expression sets the visibility of the image control if the user has chosen to include the company logo and if the conversion from binary format into text returns values greater than 0 bytes (in short, if there is image data in the dataset).

4. Click **OK** to confirm the modifications you have made.
5. Set the image control's location and size as follows:

```
Location: 3,29875in; 0in
Size: 2,42708in; 0,56578in
```

The final report layout should now look like this:

[&BCReportName] [COMPANYNAME]					[&ExecutionTime]
					[&PageNumber]
					[&UserID]




[&ItemNo_	[&PostingDa	[&CustCatPKT_	[&DocumentNo_It	[&Description_ ItemL	[&LocationCode	[&Quantity
LedgerEntry]						
		emLedgerEntry]				
LedgerEntry]	LedgerEntry]	emLedgerEntry]	_ItemLedgerEntry]	on_ ItemLedgerEntry]	emLedgerEntry]	LedgerEntry]
						gerEntry))]

Figure 8.15: Report layout

The RDL report and its layout are now ready to be deployed. Just hit *F5* and deploy the package into your sandbox.

Once the client is loaded, just search (*Alt + Q*) for the **Item Ledger Entry Analysis** report and fill in the request page if you want to include the company logo in the report's output:

Item Ledger Entry Analysis



Printer

(Handled by the browser)

Report Layout

LayoutRDL

...

Options

Include company logo

☒

Figure 8.16: Report request page

If you click **Preview & Close**, a sample output should look as follows:

### Item Ledger Entry Analysis

CRONUS International Ltd.



2/2/2023  
Page 1  
DT

Item No.	Posting Date	Customer Category	Document No.	Description	Location Code	Quantity
1000						
		GOLD				
1000	9/6/2024	GOLD	1011001			5
1000	9/7/2024	GOLD	1011002			27
						32
1100						
		BRONZE				
1100	6/1/2023	BRONZE	START-MANF			200
1100	9/7/2024	BRONZE	1011002			-27
1100	9/8/2024	BRONZE	1011003			-16
		SILVER				
1100	9/6/2024	SILVER	1011001			-5
						152

Figure 8.17: Report sample output



Values might differ depending on database version and localization. The complete object solution can be found at: <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition/tree/main/CH08/PacktDemoExtension>.

## Part 6 – Adding a Word layout

In this section, together with the RDL layout, we will also edit and design the Word layout of our report:

1. In Visual Studio Code, right-click the `ItemLedgerEntryAnalysis.report.docx` file and choose **Open externally**. This will open Microsoft Word.

2. If you have not already, be sure that you have the **Developer** tab enabled in Microsoft Word: click on **File | Options | Customize Ribbon**. In the main tab, check the **Developer (Custom)** ribbon option and click **OK**:

Word Options

?

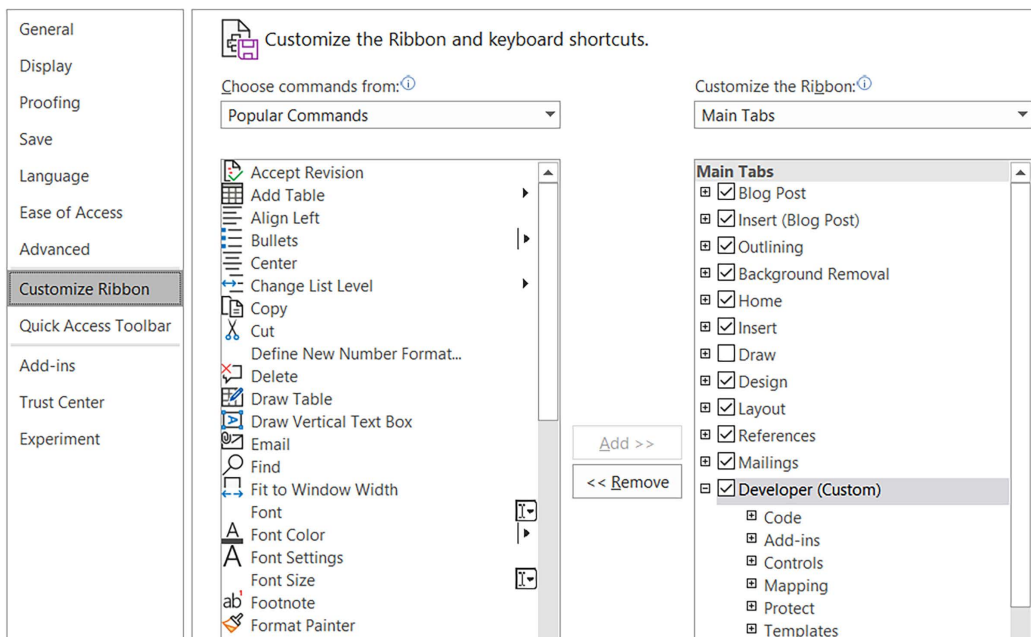


Figure 8.18: Adding the Developer tab to Word

3. Back in the Word layout, select the **Developer** tab and click on the **XML Mapping** pane.
4. In the **Custom XML Part** box, select the last entry from the drop-down menu and expand the **Labels** and **Item\_Ledger\_Entry** nodes from the **NavWorldReportXMLPart** root.

It should look like this:

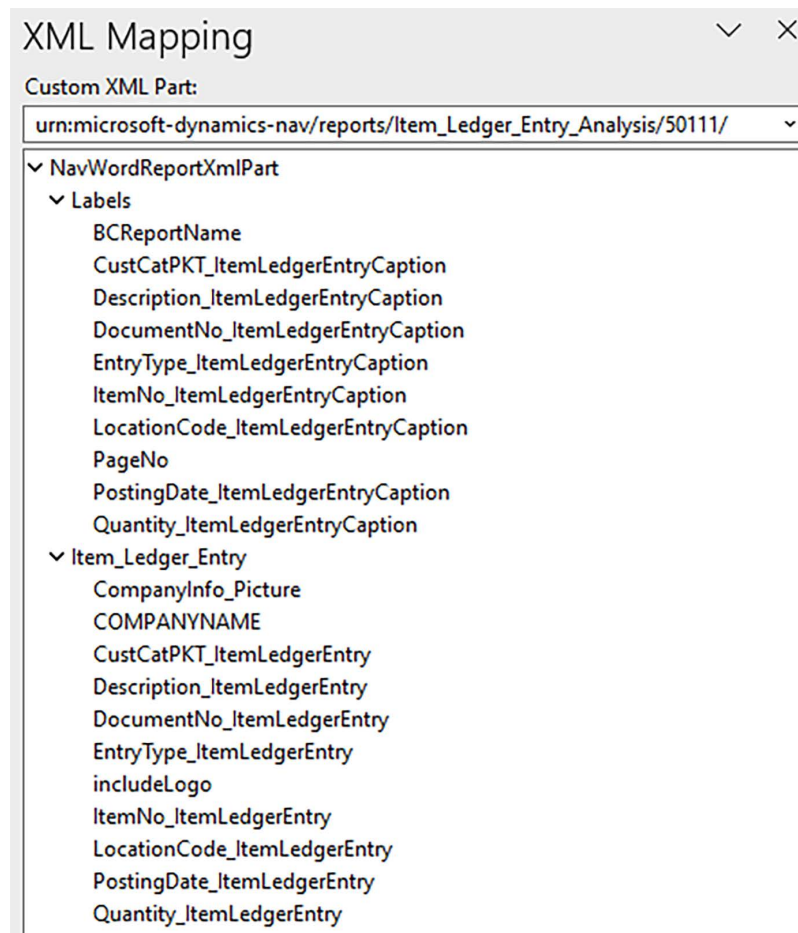
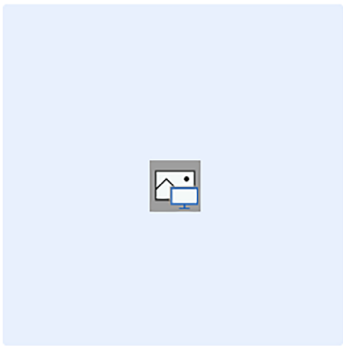


Figure 8.19: Report XML mapping

5. Let us add a Word layout list report to our extension:
6. Right-click in the **XML Mapping** pane, select **Item\_Ledger\_Entry** in the drop-down menu, and choose **CompanyInformation\_Picture**. Right-click on it and select **Insert Content Control | Picture**. This will add a placeholder for the company information logo in the Word layout.  
Add an extra line. Right-click in the **XML Mapping** pane, select **Labels**, and choose **BCReportName**. Right-click on it and select **Insert Content Control | Plain Text**.
7. Add an extra line. Right-click in the **XML Mapping** pane, select the **Item\_Ledger\_Entry** drop-down menu, and choose **COMPANYNAME**. Right-click on it and select **Insert Content Control | Plain Text**.

8. Add an extra line. In the Word ribbon, click on **Insert | Table** and create a 7-columns x 2-rows table.
9. In the first row, for each cell, place the cursor in the **XML Mapping** pane, select **Labels**, right-click in each caption element, and select **Insert Content Control | Plain Text**. Here is a list of the column captions:
  - **ItemNo\_ItemLedgerEntryCaption**
  - **PostingDate\_ItemLedgerEntryCaption**
  - **CustCatPKT\_ItemLedgerEntryCaption**
  - **DocumentNo\_ItemLedgerEntryCaption**
  - **Description\_ItemLedgerEntryCaption**
  - **LocationCode\_ItemLedgerEntryCaption**
  - **Quantity\_ItemLedgerEntryCaption**
10. Select all cells of the first row and change the font to **bold**.
11. Select the second row of the table. In the **XML Mapping** pane, select **Item\_Ledger\_Entry element**, right-click on it, and select **Insert Content Control | Repeating**. This will make the line elements repeat for every record in the Item Ledger Entry dataset.
12. In the second row, inside the repeater element, place the cursor in every cell in the **XML Mapping** pane, expand the **Item\_Ledger\_Entry** drop-down menu, right-click in the field element, and select **Insert Content Control | Plain Text**. Here is a list of the column fields:
  - **ItemNo\_ItemLedgerEntry**
  - **PostingDate\_ItemLedgerEntry**
  - **CustCatPKT\_ItemLedgerEntry**
  - **DocumentNo\_ItemLedgerEntry**
  - **Description\_ItemLedgerEntry**
  - **LocationCode\_ItemLedgerEntry**
  - **Quantity\_ItemLedgerEntry**

This will be the result:



BCReportName

COMPANYNAME

ItemNo_It mLedgerEn tryCaption	PostingDate _ItemLedger EntryCaptio n	CustCatPKT_ ItemLedgerE ntryCaption	DocumentNo _ItemLedger EntryCaption	Description_ ItemLedgerE ntryCaption	LocationCod e_ItemLedge rEntryCaptio n	Quantity_It emLedgerE ntryCaptio n
ItemNo_It mLedgerEn try	PostingDate _ItemLedger Entry	CustCatPKT_ ItemLedgerE ntry	DocumentNo _ItemLedger Entry	Description_ ItemLedgerE ntry	LocationCode _ItemLedger Entry	Quantity_It emLedgerE ntry

Figure 8.20: Report layout in Word

Save and close the Word file.

The Word layout is now ready to be deployed. Build the extension (*Ctrl + Shift + B*) and deploy the package. Once the web client is loading, search (*Alt + Q*) for the **Item Ledger Entry Analysis** report. On the request page, change the report layout to **LayoutWord** and choose to include the company logo in the report output.

Item Ledger Entry Analysis

Printer

(Handled by the browser)

Report Layout


LayoutWord

Options

Include company logo

Figure 8.21: Including a logo in a Word report

Here is a sample outcome:



Item Ledger Entry Analysis

CRONUS International Ltd.

Item No.	Posting Date	Customer Category	Document No.	Description	Location Code	Quantity
1100	6/1/2023 12:00:00 AM	BRONZE	START-MANF			200
1110	6/1/2023 12:00:00 AM	BRONZE	START-MANF			400
1120	6/1/2023 12:00:00 AM	DEFAULT	START-MANF			10,000
1150	6/1/2023 12:00:00 AM	GOLD	START-MANF			200

Figure 8.22: Report in Word with company logo

## Part 7 – Adding an Excel layout

In this section, we will also add a simple Excel layout to our report:

1. In Visual Studio Code, right-click on `ItemLedgerEntryAnalysis.report.xlsx`, and choose **Open Externally**. Microsoft Excel will open the file, and it should look like the following:

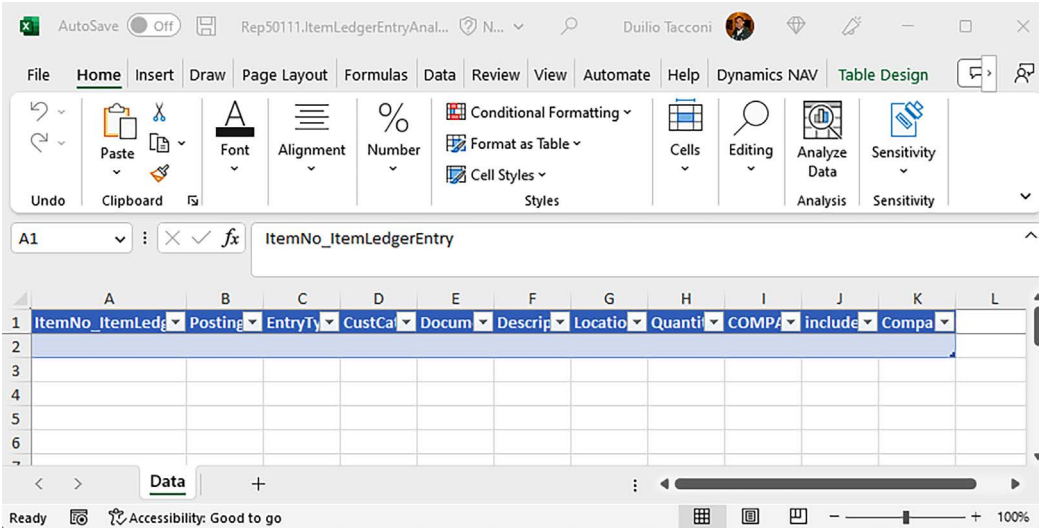


Figure 8.23: Report Excel layout

- Click on **Insert | PivotTable**, type Data in the **Table/Range** field, and choose **New Worksheet**:

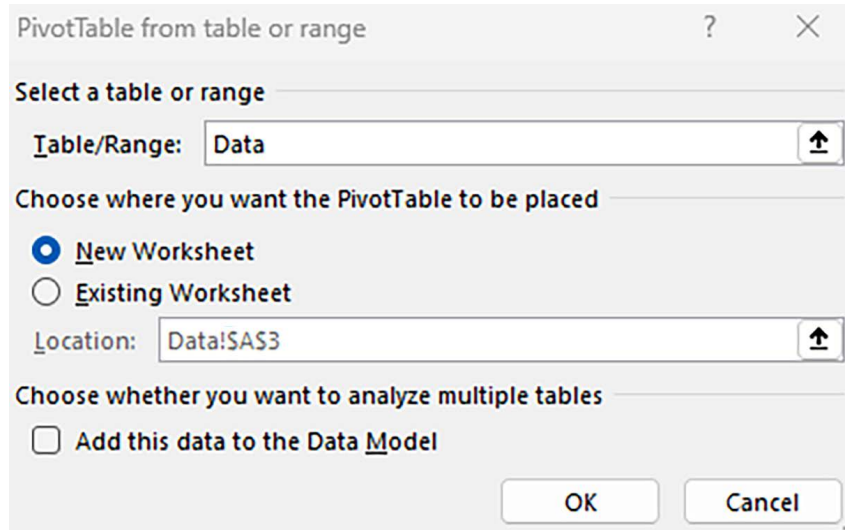


Figure 8.24: Creating a new Excel worksheet

- Rename the worksheet, when created, to **PivotTable**:



Figure 8.25: Renaming the new worksheet

- Drag and drop these fields in the respective areas:
  - CustCatPKT\_ItemLedgerEntry** : Columns
  - ItemNo\_ItemLedgerEntry** : Rows
  - Quantity\_ItemLedgerEntry** : Values
  - LocationCode\_ItemLedgerEntry** : Filters

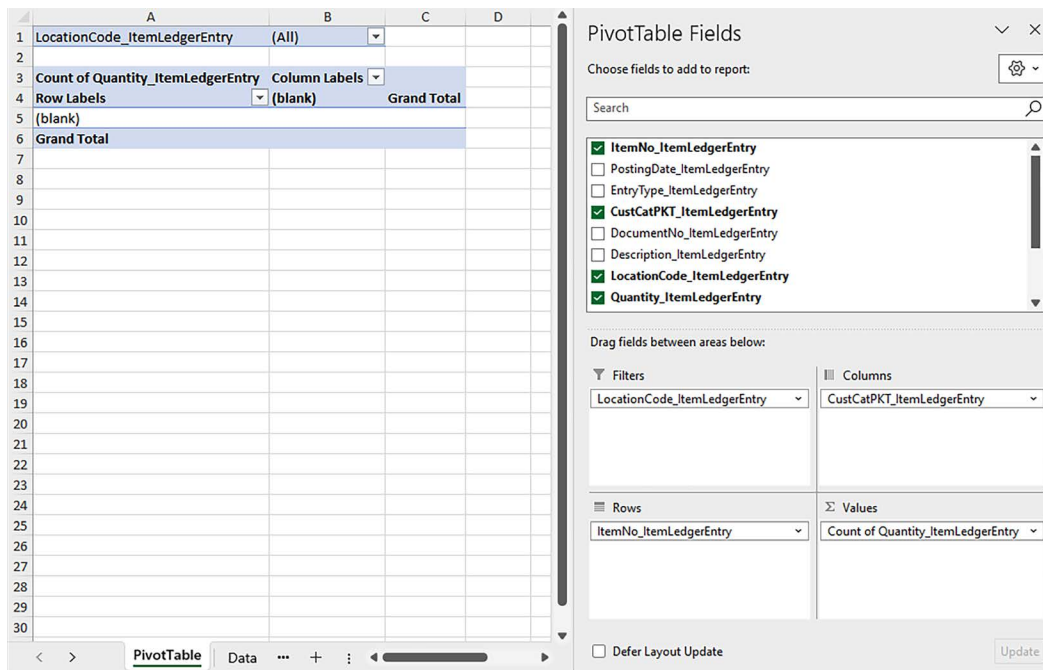


Figure 8.26: Configuring PivotTable fields

5. Expand **Count of Quantity\_ItemLedgerEntry** using the down arrow, select **Value Field Settings**, and select **summarize value field by: Sum**.
6. Then select **Number Format** and choose the **Custom** category. In the **Type** field, add: `#,##0;[Red]-#,##0`

This custom formula will export quantity values as integers, with thousands separators, and negative numbers will be displayed in red.

7. Expand every field in the area by using the down arrow, select **Field Settings | Custom Name**, and change it as follows:
  - **CustCatPKT\_ItemLedgerEntry:** Customer Category
  - **ItemNo\_ItemLedgerEntry:** Item No.
  - **Sum of Quantity\_ItemLedgerEntry:** (add a blank space)
  - **LocationCode\_ItemLedgerEntry:** Location Code
8. Now add some colors and some borders (or whatever color and border format you prefer) between cells A1 and C6. In the end, it should look like the following:

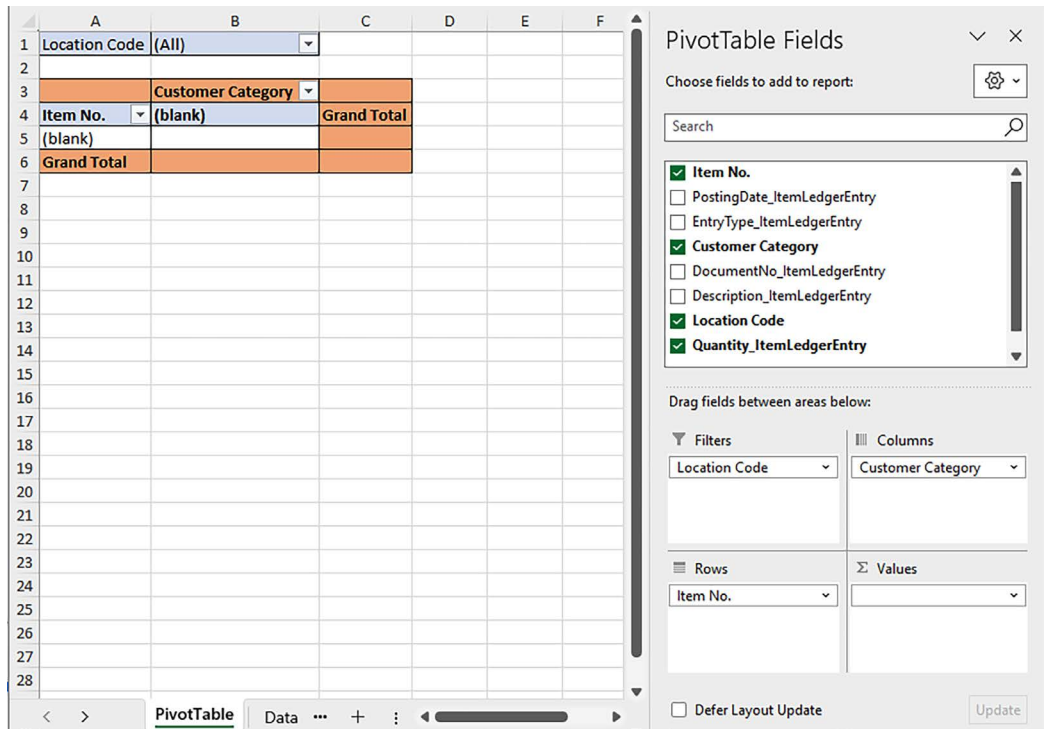


Figure 8.27: Formatting report cells in Excel

9. For one last touch of maquillage, click on **Insert | Recommended Charts**. Change **Clustered Column** to **3-D Column**. In the graph, change the title to **Item categories by Location**.

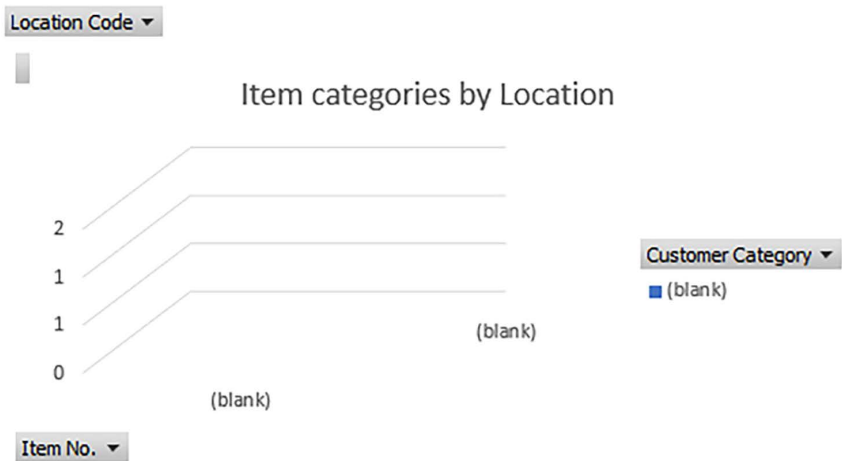


Figure 8.28: Adding a chart to the Excel report

Save and close the Excel file.

Build the extension (*Ctrl + Shift + B*) and deploy the package. Once the web client is loading, search (*Alt + Q*) for the **Item Ledger Entry Analysis** report. On the request page, change the report layout to LayoutExcel:

## Item Ledger Entry Analysis



Printer ..... (Handled by the browser) ▼

Report Layout ..... LayoutExcel ...

Figure 8.29: Report Layout option for Excel

Click on **Download**. Here is a sample outcome:

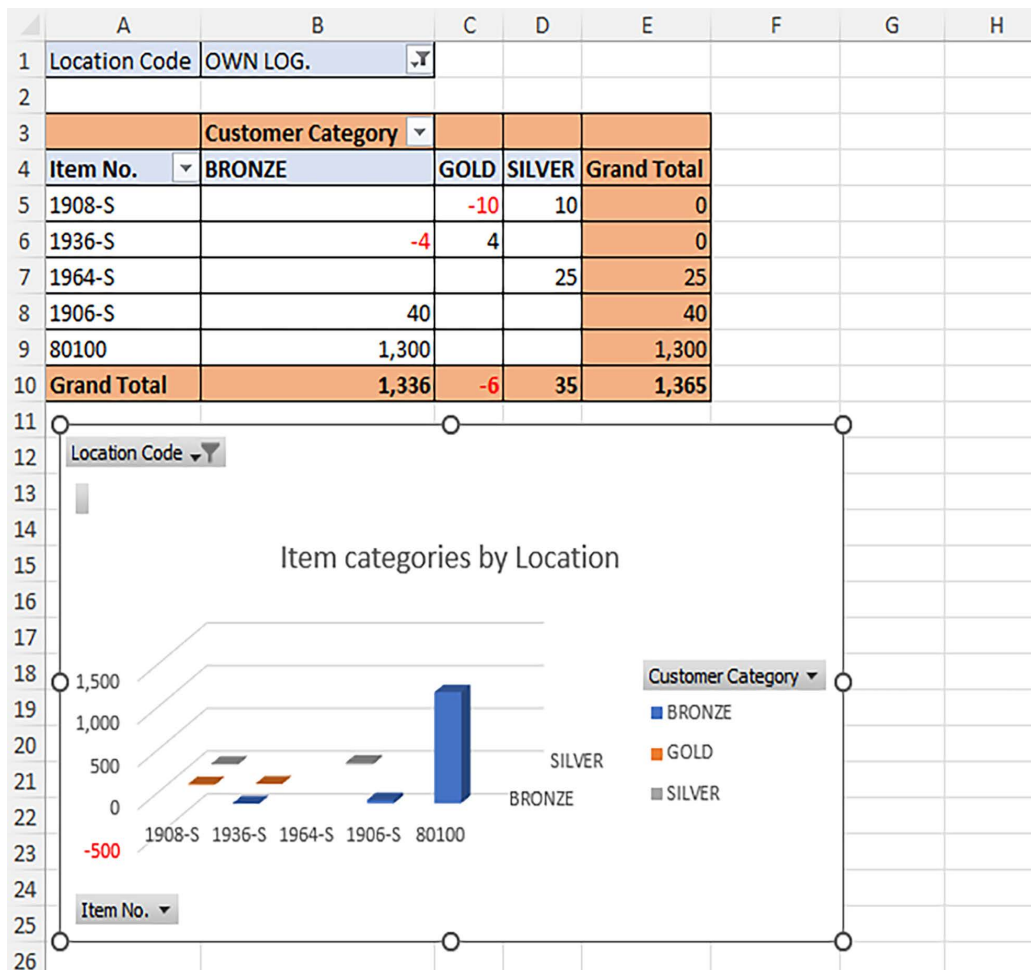


Figure 8.30: Final report layout in Excel

That concludes this section, where we created and beautified a report from scratch. In the next section, we will have a look at the most common tasks for a developer: copying and refactoring an existing report to cope with customer feature requirements, or using a report extension object to extend a standard or third-party report.

## Report extension object: a basic example

Making small modifications to an existing report is a quite common task. Together with creating new pages and codeunits, this is one of the most repetitive and frequent jobs for a developer.

Let us say that we would like to make the following changes to a standard sales order report:

- Show the Customer Category field in the sales order header.
- Print *GIFT* labels in the sales lines for item lines with a 100% discount.

The very first question that a developer must ask is: would this be suitable to be done through a ReportExtension object? Depending on the situation, and on your preferences, you will choose between extending a report and cloning a report. We will learn more about cloning in the next section.

For now, let's say that we are using a report extension object. The benefit of this is that we can delegate maintenance of the core business logic to the author of the source (target or extended) report. In this case, we just need to:

- a. Add columns to an existing dataset.
- b. Find out the appropriate event publishers in the sales order report.
- c. Subscribe to these publishers as needed.
- d. Create a copy of the layout and make the necessary changes in the header and lines.

Let us do this together, step by step:

1. Create a new folder called `reportextension` in the `.\Src\Gifts` folder.



Once you're comfortable using extensions, a convenient practice is to use **Waldo's AL extension**, which will automatically name your objects, and you can let it organize them too. This will create the object-type folders for you.

2. Create a new file in the newly created folder called `SalesOrderConfExt.reporttext.al`.
3. Type `reporttext` to enable the report extension snippet.
4. Add or change the report extension as follows:

```
reportextension 50111 SalesOrderConfExt extends "Standard Sales - Order
Conf."
{
    dataset
    {
        modify(Header)
```

```

    {
        trigger OnAfterAfterGetRecord()
        begin
            Customer.get(Header."Bill-to Customer No.");
        end;
    }

    add(Header)
    {
        column(CustomerCategory_PKT;Customer."PKT Customer Category
Code") { }
        column(CustomerCategory_PKT_Lbl;Customer.FIELDCAPTION("PKT
Customer Category Code")) { }
        column(GiftLbl;GiftLbl) { }
    }

    modify(Line)
    {
        trigger OnAfterAfterGetRecord()
        begin
            case "Line Discount %" of
                0      : NewLineDiscountPctText := '';
                100    : NewLineDiscountPctText := GiftLbl;
            else
                NewLineDiscountPctText := StrSubstNo('%1%',-
Round("Line Discount %",0.1));
            end;
        end;
    }

    add(Line)
    {
        column(NewLineDiscountPctText;NewLineDiscountPctText) { }
    }
}

rendering
{
    layout(LayoutWord)
    {
        Type = Word;
        Caption = 'PKT Standard Sales - Order Conf.';
    }
}

```

```
        Summary = 'Standard Sales - Order Conf. report ext.';
        LayoutFile = '.\Src\Gifts\reportextension\SalesOrderConfExt.
reporttext.docx';
    }
}

var
    Customer : Record Customer;
    newLineDiscountPctText : Text;
    GiftLbl : Label 'GIFT';
}
```

Note that you can find this object in this book's code repository at <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition/tree/main/CH05/PacktDemoExtension>

- 5. Run the web client and search for the Report Layouts list. Filter by **Report ID 1305** and **Type Word**; click on the **Export Layout** action:

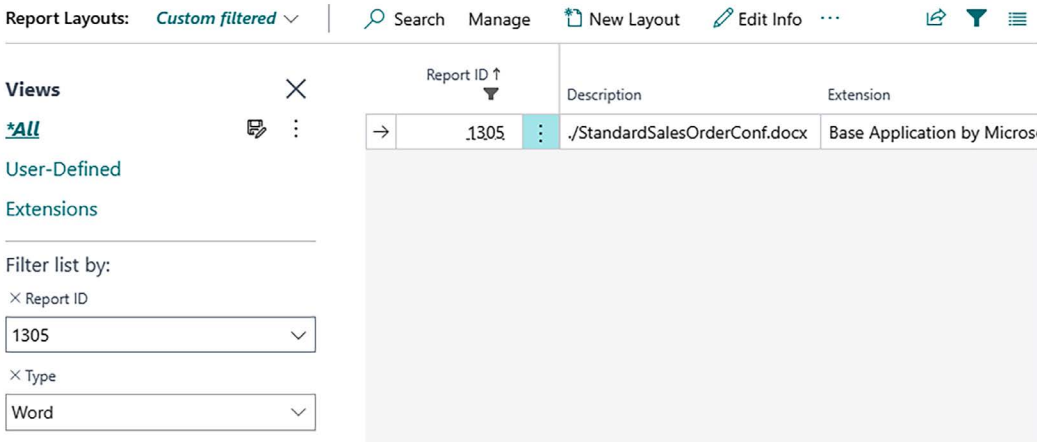


Figure 8.31: Finding the Export Layout action

- 6. Copy and rename (substitute) the file download into .\Src\Gifts\reportextension\SalesOrderConfExt.reporttext.docx.
- 7. Compile the extension (*Ctrl + Shift + B*). This action will add the new dataset columns in the Word layout file.
- 8. Right-click on the SalesOrderConfExt.reporttext.docx file and choose the **Open Externally** option.
- 9. In the Word layout, remove the CustomerAddress8 field and instead, add the following:
  - CustomerCategory\_PKT\_Lbl as plaintext
  - A space, a colon (:), and another space
  - CustomerCategory\_PKT as plaintext


This is how it should appear:

CustomerAddress1  
CustomerAddress2  
CustomerAddress3  
CustomerAddress4  
CustomerAddress5  
CustomerAddress6  
CustomerAddress7  
CustomerCategory\_PKT\_Lbl : CustomerCategory\_PKT

Figure 8.32: Replacing the CustomerAddress8 field

- 10. Locate and select LineDiscountPercentTextLine in the repeater line and delete it.
- 11. In the same place where it was located, place the cursor and from the **XML Mapping** pane select **NewLineDiscountPercentText**, and add it as plaintext.
- 12. Save the Word layout file, compile the extension (*Ctrl + Shift + B*), and publish it.

Now, run any sales order that has a gift line and customer category assigned and, on the request page, change the layout to **PKT Standard Sales - Order Conf.**



You can make this your default layout for this report by simply selecting the report and clicking on the **Set Default** action on the **Report Layouts** lookup page.

Report Layouts | 🔍 ▼ ⋮

Report ID ↑ ▼		Layout Name
1305		./StandardSalesOrderConf.rdlc
1305		./StandardSalesOrderConf.docx
→ 1305	<div><div>Manage &gt;</div><div><div>New Layout</div><div>Edit Info</div><div>Run Report</div><div><b>Set Default</b></div><div>Export Layout</div><div>Replace Layout</div><div>✓ Show as menu</div></div></div>	<b>PKT Standard Sales - Order Conf.</b>

Figure 8.33: Setting the sales order layout

Inspecting the result, it should look like this:

order confirmation

101005

January 11, 2024

Page 1 / 1

John Haddock Insurance Co.

Miss Patricia Doyle

10 High Tower Green

Manchester, M02 4RT

Great Britain

CRONUS International Ltd.

5 The Ring

Westminster

W2 8HG London

Customer Category Code : SILVER

External Document No.

Salesperson

Quote No.

Shipment Method

Jim Olive

Ex Warehouse

No.	Description	Quantity	Unit Price Excl. VAT	VAT %	Line Amount Excl. VAT
1920-S	ANTWERP Conference Table	4	Piece 420.40	GIFT 25	0.00
Total GBP Incl. VAT					0.00

Figure 8.34: Sales order layout

## Cloning and refactoring reports

Cloning reports is a state-of-the-art feature. That said, we should stop for 10 seconds and think “am I doing the right thing?”.

For complex or radical changes within a report, it is required to start from scratch or, more frequently, to clone and refactor an existing report. In fact, even for simple things like adding a field to an existing report, you could consider cloning a report. The report is much easier to deploy that way. You may end up preferring cloning reports over report extensions in most cases. However, you will have a more comprehensive knowledge of report objects if you understand report extensions, hence us waiting till now to cover cloning.

The first task when cloning a report is straightforward since AL source code is available in various places and easily downloadable.

With the on-premises version, you can find the report you want to clone in the DVD installation folder in the \Applications\BaseApp\Source directory. Just unzip the file named Base Application.Source.zip and search for the standard report you want to clone.

The very same source file can be downloaded from online repositories using the BcContainerHelper PowerShell library and using the command `Get-BcArtifactUrl` to get the URL of the desired online or on-premises build. Then simply use the `Download-Artifacts` cmdlet fed with the URL, and after the download is complete, you can find the downloaded file in the `c:\bcartifacts.cache` folder.

Another way to download the source code is to search for the specified object in the following well-known MVP repository: <https://github.com/StefanMaron/MSDyn365BC.Code.History>.

Let us do it together with a practical example.

Imagine that you have been asked to add a QR code to a sales invoice header that would store the following comma-separated values: invoice no., document date, customer category, all VAT totals, and total amount including VAT.

This is a good example because it prompts the question: why not use a report extension object instead? The answer is simple: in this example, we need to access variables from the report. These are not exposed in any way to a report extension object, so we only have the options of cloning and customizing the report or creating a brand-new one from scratch.

Here is all you need to do:

1. Create a new file called `PacktSalesInvoice.report.al` in the `.\Src\CustomerCategory\report` folder.
2. Go to the <https://github.com/StefanMaron/MSDyn365BC.Code.History> repository and search for `StandardSalesInvoice.Report.al`.
3. Once the Sales Invoice report is shown, click on **Copy raw contents**.
4. Paste the raw content you just copied into the file created at point 1 and change the report properties as follows:

```
report 50116 "Packt Sales - Invoice"
{
    WordLayout = '.\Src\CustomerCategory\report\PacktSalesInvoice.report.docx';
    Caption = 'Packt Sales - Invoice';
}
```

5. Run the client, go to **Report Layouts**, and filter for the report ID 1306 and Type Word:

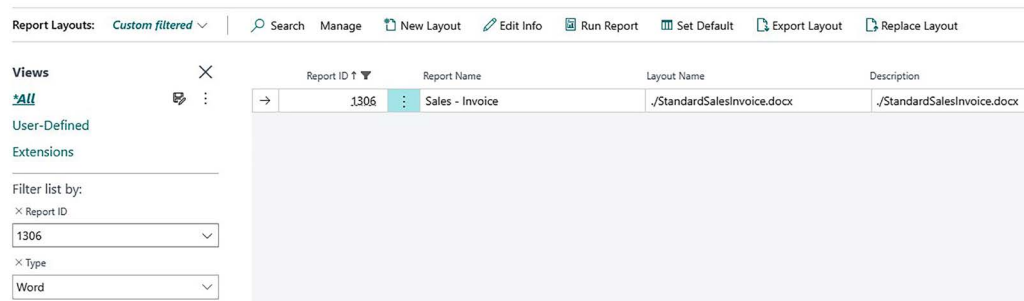


Figure 8.35: Finding the Word report layout

6. Use the **Export Layout** action to export the docx layout.
7. Move and rename the layout according to the report's properties. In our example, it should look like this: `.\Src\CustomerCategory\report\PacktSalesInvoice.report.docx`
8. That concludes the cloning operation. Now it is time to add our custom code.
9. Add two report global variables called `PKTBarcodeString` and `PKTEncodedString` as Text.
10. Initialize the variable in the `OnAfterGetRecord` trigger in the Header data item:

```
trigger OnAfterGetRecord()
var
    CurrencyExchangeRate: Record "Currency Exchange Rate";
    PaymentServiceSetup: Record "Payment Service Setup";
    Currency: Record Currency;
    GeneralLedgerSetup: Record "General Ledger Setup";
begin
    PKTBarcodeString := ''; //add this line
    PKTEncodedString := ''; //add this line
```

11. Recursively calculate and add the VAT totals in the `ReportTotalsLine` data item `OnAfterGetRecord` trigger:

```
trigger OnAfterGetRecord()
var
    Customer : Record Customer;
    CustomerCategory : Code[20];
begin
    if PKTBarcodeString = '' then begin
        if Customer.get(Header."Bill-to Customer No.") then
            CustomerCategory := Customer."PKT Customer Category Code"
        else
            CustomerCategory := '';

        PKTBarcodeString := format(Header."No.") + ' ' +
            format(Header."Document Date") + ' ' +
            CustomerCategory;
    end;

    PKTBarcodeString += '[' + Description + ' ' +
        format(Amount, 0,
            AutoFormat.ResolveAutoFormat("Auto Format"::AmountFormat,
            Header."Currency Code")) + ']';
end;
```

12. Encode the barcode string in the Totals data item OnAfterGetRecord trigger:

```

trigger OnAfterGetRecord()
var
    BarcodeSymbology: Enum "Barcode Symbology 2D";
    BarcodeFontProvider : Interface "Barcode Font Provider 2D";
    Customer : Record Customer;
    CustomerCategory : code[20];
begin
    if PKTBarcodeString = '' then begin
        if Customer.get(Header."Bill-to Customer No.") then
            CustomerCategory := Customer."PKT Customer Category Code"
        else
            CustomerCategory := '';

        PKTBarcodeString := format(Header."No.") + ' ' +
            format(Header."Document Date") + ' ' +
            CustomerCategory + ' ';
    end;

    PKTBarcodeString += format(TotalAmountVAT, 0,
        AutoFormat.ResolveAutoFormat("Auto Format"::AmountFormat,
            Header."Currency Code")) + ' ' +
        format(TotalAmountInclVAT, 0,
            AutoFormat.ResolveAutoFormat("Auto Format"::AmountFormat,
                Header."Currency Code"));

    BarcodeFontProvider := Enum::"Barcode Font Provider
2D"::IDAutomation2D;
    BarcodeSymbology := Enum::"Barcode Symbology 2D"::"QR-Code";
    PKTEncodedString := BarcodeFontProvider.EncodeFont(PKTBarcodeString,
BarcodeSymbology);
end;

```

13. Add the following column to the Totals data item:

```

column(PKTEncodedString;PKTEncodedString)
{}

```

14. Compile the extension (*Ctrl + Shift + B*) to push the new column just added in the Word layout. Now, let us change the layout to print the QR code.
15. Right-click `.\Src\CustomerCategory\report\PacktSalesInvoice.report.docx` and choose **Open Externally**.

16. Position the cursor just before the lines and after the **Work Description** field, or wherever the QR code will best fit in the layout for you, and select PKTEncodedString as plaintext from the list of fields in the **XML Mapping** pane.
17. And now for the important last part: change the font family to IDAutomation2D and the font size to 10, as shown below for the PKTEncodedString field:

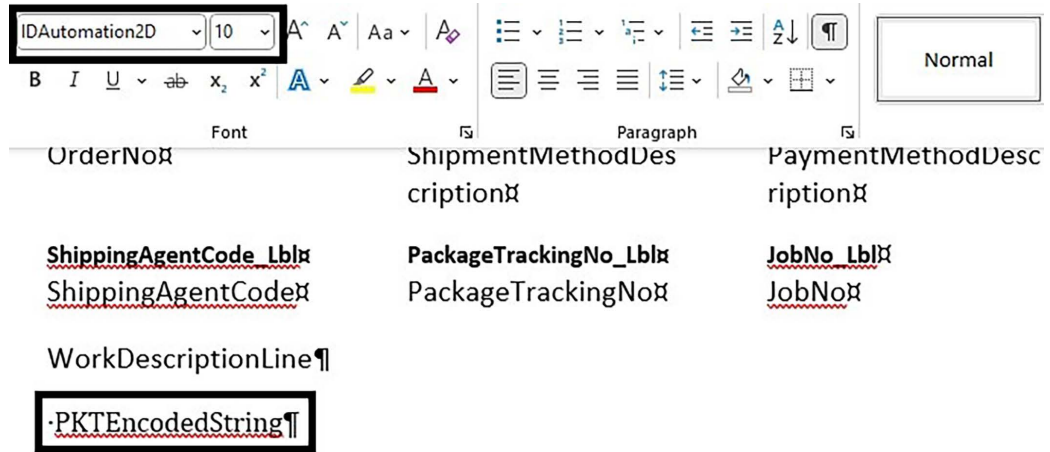


Figure 8.36: Configuring the font family

This will complete the layout and print the QR code on every document.

To make it more effective, we must let the application “understand” that every time standard report 1306 is invoked, it should be substituted with custom report 50116 instead. This is easily done by subscribing to a specific event publisher in the ReportManagement codeunit named OnAfterSubstituteReport.

18. Edit `.\Src\CustomerCategory\report\PacktSalesInvoice.report.docx` and add the following code:

```
[EventSubscriber(ObjectType::Codeunit, Codeunit::ReportManagement,
'OnAfterSubstituteReport', '', false, false)]
local procedure OnAfterSubstituteReport(ReportId: Integer; var
NewReportId: Integer)
begin
    if ReportId = Report::"Standard Sales - Invoice" then
        NewReportId := Report::"Packt Sales - Invoice";
end;
```

19. Build (*Ctrl + Shift + B*) and publish (*Ctrl + F5*) the extension. Now the whole (code and layouts) standard sales invoice report will automatically be replaced behind the scenes with the custom one. Below is some sample output:

Your Reference	Salesperson	Due Date	Payment Terms
	Jim Olive	April 30, 2023	Current Month
Order No.	Shipment Method	Payment Method	
Shipping Agent Code	Package Tracking No.		



Figure 8.37: Sample invoice output

That concludes our section related to cloning, customizing, and refactoring existing AL reports. Next, let us learn how feature limitation works on the most-used report layouts for Dynamics 365 Business Central document reports: RDL and Word.

## Feature limitations when developing RDL or Word layout document reports

Generally, professional report development should be done by developing RDL report layouts with Visual Studio and the RDLC report extension installed. The Word document layout has more limitations than RDL, and its main advantage is that it is quite popular and easy to adopt by power users.

The major pain points you might find when developing layouts are typically related to documents. The most well-known ones are as follows:

- **Header and footer space is always retained:** Report headers and footers have static content, and they have been engineered so that they are always displayed on every page if they are present. Nevertheless, with RDL, you could use the typical `SetData` function in the body and the `GetData` function in the header.
- **There's no easy way to mimic the `PlaceInBottom` property for a document report from the old classic client report:** When developing a document, you might be asked to generate an entire document, but the totals (VAT, totals per group, and so on) must be always printed at the bottom, always in the same place, and only on the last page. Here, a problem arises because a Dynamics 365 Business Central document report could be considered as *a batch of multiple documents and multiple copies* rather than a single report. This implies that the breakdown for renumbering pages must be done for every document number and copy number.

- In standard document reports, totals are never placed at the bottom of the page; they are printed right after the last document line. This means that they could be printed anywhere on a page, or even on an extra page. There is no feasible way to print a batch of the document and have the totals printed at the bottom of the last page for every document and copy.
- **Complexity in implementing running totals:** Typically, with documents, you would like to implement running totals at the bottom of the page that are reported at the top of the next page, such as **To be Continued** and **Continue** labels when printing the ledger or some transactional entries.
- With the old classic client report designer, you could resolve this by adding a transheader/transfooter. These artifacts no longer exist with RDL or Word layout reports. With a Word layout report, there is no feasible solution. With RDL, you might implement running totals, but only with header and footer sections. This is an old but useful development reference: <https://blogs.msdn.microsoft.com/nav/2011/06/06/transfooter-and-transheader-functionality-in-rdlcssrs-reports-revisited/>.

Considering Word layouts only, the design limitations you might also frequently hit are as follows:

- **No conditional formatting:** If you need to set the visibility of a control, change a field value within the layout, or set any conditional formatting, this is not possible with Word. A typical example is when you need to print a blank character instead of zero. This must be done in the dataset, and values must be sent to the document already formatted as strings (blank or with a numeric value).
- **No totaling formulas:** There is no equivalent to RDL `=SUM` functions or similar. Value calculations must be done through AL code and the result is added to the dataset.
- **Having nested repeaters in the same table is a challenge:** Since it is not possible to conditionally trigger the visibility of a table line, there is a tradeoff between having a good-looking layout and having different data items in the same table. A typical example is commenting lines under a sales line or an additional description/barcode line under a sales shipment line.
- These might be achieved by adding a nested table structure repeater within the line's repeater, which maps the extra additional information. The nested table structure can be freely defined, if it spans a set of merged cells in the outer table. When you develop this, be aware that if there is no extra additional information, at least one empty instance of the nested structure will be included. The best solution would be to use a buffer table in the dataset and create the exact line structure as it must be printed in the layout.

If you come across one or more of these limitations, then probably the best and easiest solution is to develop an RDL layout report instead.

## Understanding report performance considerations

With Dynamics 365 Business Central online, there are performance considerations that need to be considered.

Currently, all built-in layouts are rendered in the same application domain process when they run with `SAVEASPDF` or `SAVEAS` statements.

Since RDL layouts might enable some external code artifacts that may potentially affect data within the same application domain, it has been decided to run every custom RDL report layout in isolated mode. It is worth noticing that if you develop a report and declare `DefaultLayout` as RDL and the `RDLLayout` property, this is considered a built-in layout and should render in the same application domain.

Word and Excel layouts, no matter whether they are built-in or custom-made, do not run in isolation.

Enabling application domain isolation for custom RDL layouts provides a more secure and reliable processing environment. However, the drawback is that it could increase the rendering time.

Whenever you develop RDL reports for Dynamics 365 Business Central online, you must also test performance with the `SAVEASPDF` or `SAVEAS` statements within an online sandbox or a Docker-contained sandbox, with the `customsettings.config` file server parameter, `ReportAppDomainIsolation`, set to `true`.

Other performance considerations that are valid for both Word and RDL report layouts are based on dataset optimization.

These are the equations to keep in mind when developing the data structure of the AL report. Let us consider that the dataset is an in-memory table (X axis = columns and Y axis = rows):

```
Smaller Dataset = Better Performance
Smaller Dataset = Reduce X axis (columns) + Reduce Y axis (rows)
Better Performance = Optimize (reduce) the number of columns in Dataset +
Optimize (reduce) the number of records processed in DataItems
```

You can use the following links to optimize standard reports or your own custom reports. They are old but still relevant, since RDL reports have not changed that much for more than a decade:

- <https://blogs.msdn.microsoft.com/nav/2014/03/09/rdlc-report-and-performance-in-microsoft-dynamics-nav/>
- <https://blogs.msdn.microsoft.com/nav/2016/05/20/rdlc-report-and-performance-in-microsoft-dynamics-nav-2015-and-2016/>
- <https://blogs.msdn.microsoft.com/nav/2015/03/17/a-couple-of-rdlc-performance-optimization-tips/>

## Summary

In this chapter, we learned about what tools to use to develop reports with AL. We saw how to create RDL, Word, and Excel layouts and what tools are supported. We got a better understanding of the creation of report and report extension objects and when to use them. We also explained how to clone an AL report and refactor it to be reused within the standard Business Central application with a practical example.

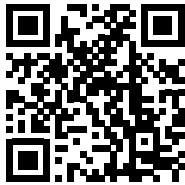
Finally, we learned that there are some reporting limitations, a few workarounds, and some performance considerations that can help you become a master of AL report development.

In the next chapter, we will learn about the basics of cloud-ready printing and what under-the-hood events to subscribe to if you want to handle your reporting output in a different way than the standard application does.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>





# 9

## Printing

Moving an ERP application to a web-based approach in the cloud implies a change of development mindset when accessing external resources. In *Chapter 7, File Handling*, we have already addressed this mindset switch, considering local or shared folders and files, and now we will also cover the different ways of approaching printers and similar devices as an external resource.



This chapter is intended to provide you with a general idea of how to approach cloud printing and define what could be the best strategy for you to take. This chapter is not intended to create a custom cloud printer service.

In the old days of the classic Windows client, local resources such as printers were directly accessible through legacy platform assemblies inspecting registry key definitions. Today, this is still valid only when printing server side on-premises, where the Dynamics 365 Business Central Server service (so-called **NAV Service Tier (NST)**) can be instructed to print through a scheduled job. The NST assemblies will inspect the server registry for the printer definitions and print out the report as defined through AL code.

Within SaaS, there is no access to a server-side Windows registry that stores any record for printers. How to manage this constraint will be demystified in this chapter, where we will cover the following topics:

- Understand the different cloud-ready printing options
- Implement Microsoft Universal Print
- Deep dive into the modern printing structure
- Alternatives to Microsoft Universal Print

# Understand cloud-ready printing

To define a printer in Dynamics 365 Business Central, the starting point is always the **Printer Management** page.

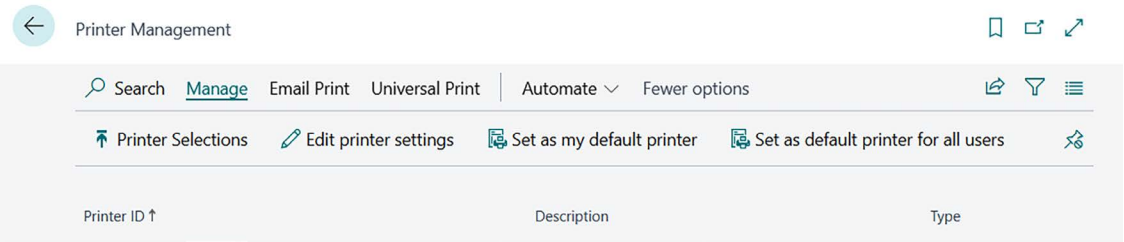


Figure 9.1: Printer Management page

This page is where you add printer devices targeting email printing services or the Microsoft Universal Print printing service. Such service types require the printers to be cloud-ready, and we will dig deeper into these shortly.

Once records have been added to the **Printer Management** page, it is possible to drill through the **Printer Selections** page and combine the **User ID**, **Report ID**, and **Printer Name** fields. Using those fields in various ways, you can configure the following settings:

Setting	Configuration method
Set a specific printer as default for all reports	Assign a printer name and leave blank both the <b>User ID</b> and <b>Report ID</b> fields or choose the <b>Set as default printer for all users</b> action from the <b>Printer Management</b> page.
Set a specific printer for a specific user	Assign a printer name and a <b>User ID</b> and leave blank the <b>Report ID</b> field or let the user choose the action <b>Set as my default printer</b> from the <b>Printer Management</b> page.
Set a specific printer for a report	Assign a printer name and a <b>Report ID</b> and leave blank the <b>User ID</b> field.
Set a specific printer for a report for a specific user	Assign values to all fields: <b>User ID</b> , <b>Report ID</b> , and <b>Printer Name</b> .

Table 9.1: Printer settings and ID fields

If there is no cloud-ready printer specified in Dynamics 365 Business Central, the application will always let the browser choose which printing action to perform when selecting the **Print** action from a report request page (**Handled by the browser**).

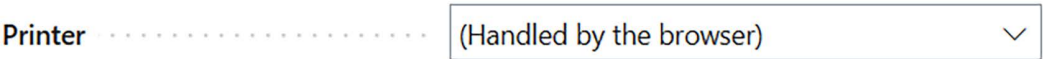


Figure 9.2: Print action chosen by the browser

Typically, the supported browsers will open their system printer dialog in preview mode with the client default printer and its parameters selected.

In most cases, users will typically preview or review what to print out to avoid wasting paper or simply check if there are imperfections. For this reason, it is today widely accepted to preview or download a PDF file first, instead of directly printing it out.

There are situations in which users would like or strictly need to print *directly* to a specific device: and this is where the problem starts. Since there is no strict interaction between the application and printers, these two worlds should find a way to communicate with each other. There are two main communication approaches to resolve this task: send the output to be printed out via email or through APIs. In both cases, we are clearly dealing with service interfaces (an email service or a web service).

## Email printers

The first version to enable email printers was Dynamics 365 Business Central 2020 Wave 1, in April 2020. At that time, Microsoft searched around for the most modern printing services, and the vast majority – if not all – were based on an email service. Roughly speaking, every major printer provider has an email printing service where a unique email account maps to an IP address for a home or industrial printer. Sending an email with a PDF attachment to that specific email address would mean contacting the email service that will process and redirect the attachment to the specified printer and start the printing job. Setting up an email printer is straightforward; you can simply follow the official documentation, prior to setting up an email account in Dynamics 365 Business Central: <https://learn.microsoft.com/en-us/dynamics365/business-central/admin-printer-setup-email>

Below is a sample of the **Email Printer Settings** card from the **Printer Management** list page:

←

Email Printer Settings

✓ Saved

HP 33B624 TANGO

New

More options

Printer ID

HP 33B624 TANGO

Description

Sends print jobs to the printer's email address

Printer Email Address

HP3B6243456623GHF456678J@hpeprint.com

Paper Size

A4 paper (210 mm by 297 mm).

▼

Landscape

☒

Email Subject

Attachment to be printed

Email Body (Optional)

Figure 9.3: Email Printer Settings sample

The two main drawbacks that you might encounter in this process are related to:

- **High processing times:** Compiling the email, sending it out from Dynamics 365 Business Central, and processing it through the external printing service could result in it taking 30-40 seconds before the printing job starts. The best advice is to test your solution before implementing it. This would be the main factor influencing your choice of an email printer or another cloud-ready printing service instead.
- **Email or attachment file size limitation:** Most email-printing service providers allow a maximum attachment dimension for processing. To give you an example, the HP ePrint service is limited to a total size of 10 MB per email. If you are printing a batch of sales orders, invoices, or whatever document, it would be worth filtering them out and printing them in smaller chunks, instead of finding out that the service refused to process it due to its operational limit per job being reached.

## Microsoft Universal Print printers

Universal Print is a Microsoft Azure service and has been designed to be a top-class cloud-based centralized printer management system. You could map any printer directly, if its driver is Universal Print-ready, or through the Universal Print connector downloaded and installed locally. It is a subscription-based service, hence you pay as much as you consume: the more you print, the more you pay. To learn more about the offering, please visit <https://www.microsoft.com/en-us/microsoft-365/windows/universal-print>

Conceptually, it is quite simple to explain:

1. Be sure your printer is Universal Print-ready or install the Universal Printer connector locally.
2. Map the printer in the online service catalog.
3. A Universal Print printer's catalog and printer jobs are exposed through APIs. An application consumes these APIs to gather printer properties and send files to be printed out in jobs.
4. The Universal Print service acts as a source for validating permissions and redirects the printing job to the specified printer.

Compared to most email printer services, Universal Print is fast since it does not need to carry out all the email preparation, send, and receive tasks. If you choose to implement this solution, you might notice that the printing job will start locally almost immediately after the report rendering has ended.

Let us analyze the whole setup process in detail. First, we will look at how to deploy the Universal Print connector.

## Universal Print connector

If your printer device does not have a printer driver that is Universal Print-aware, then you simply must download and install the latest and greatest version of the Universal Print connector locally to establish a connection between the Azure printer catalog and the device.

To successfully deploy the connector, there are some prerequisites to be fulfilled:

1. The installation host must be **Windows 10 64-bit, Pro or Enterprise**, version **1809** or *later*, or **Windows Server 2016 64-bit** or later.
2. The installation host must have an internet connection.
3. **.NET Framework 4.7.2** or *later* must be installed.
4. **Universal Print** must be enabled in Azure.
5. The connecting user account must have the role **Printer Administrator** or **Global Administrator** in Azure.
6. The connecting user account must have a Universal Print license assigned in Microsoft 365.

The official download link is <https://aka.ms/UPConnector> and installing locally is super easy. Just accept the license term and it will be deployed in a few seconds:

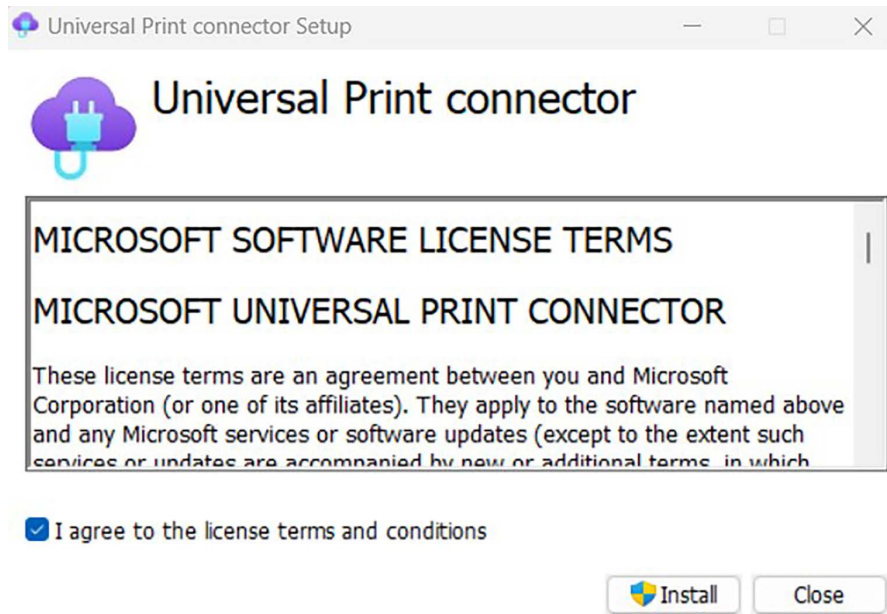


Figure 9.4: Installing the Universal Print connector

Once deployed, you might notice a new service running called Print Connector service:

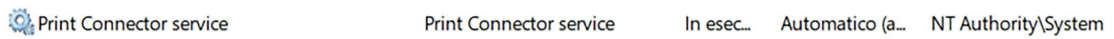


Figure 9.5: Print Connector service

To finalize the configuration, it is required that you sign in with the connection service user and give the connector service a nickname, like, for example, UP + PC name, and register it in the Azure catalog through the connector service.

## Universal Print portal

Once you have registered your connector, connect to Azure, search for Universal Print, and click on it: this will open the Universal Print portal. By clicking on **Connectors**, you should be able to see the record inserted by your deployment. In the example below, there is one called **UP-PCTaconi**:

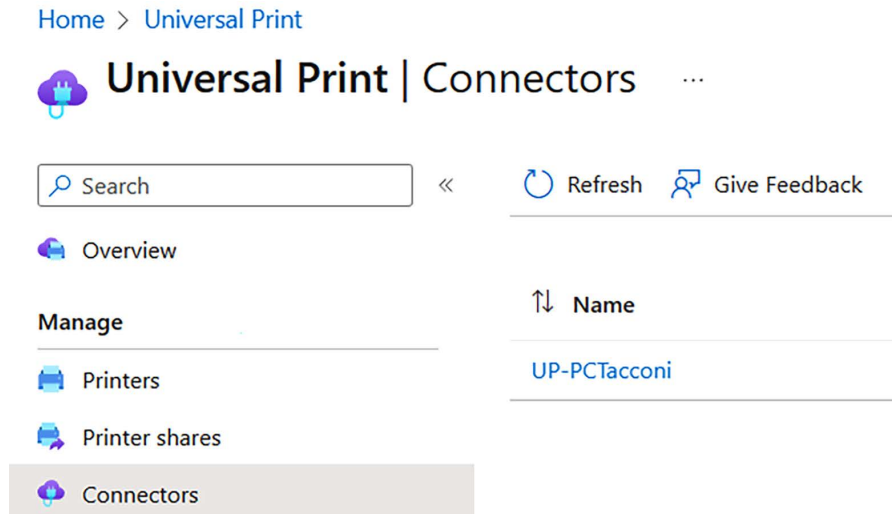


Figure 9.6: Universal Print Connectors list

Go back to the local connector instance, select the printers that you would like to add to the Universal Print catalog, and click on **Register**. These printers will be added to the online list as shown below and listed among the registered printers by the local connector:

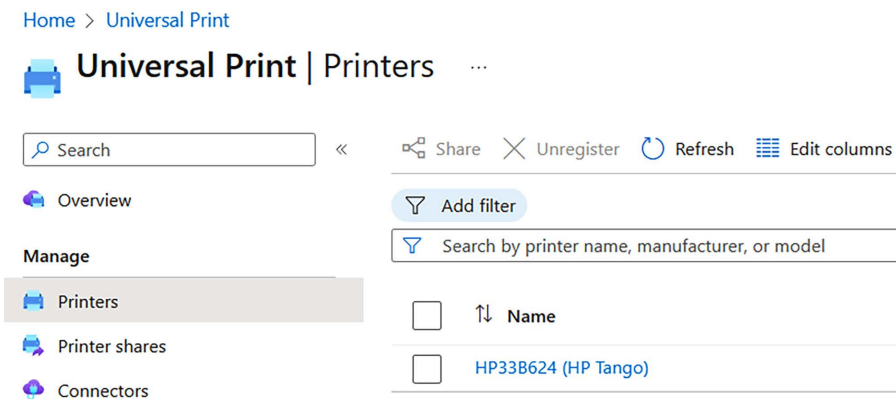


Figure 9.7: Universal Print Printers list

Clicking on the printer name will drill you through the printer card, where it is possible to inspect general info, such as model and manufacturer, but also manage printer jobs, properties, and connectors.

When submitting a printer to be used with Dynamics 365 Business Central, the most important property to be correctly set is **Content type**. To change it, drill through the printer name and click on

**Properties**; then choose the **Printer defaults** tab, set **Content type** to **application/pdf** (the default is **application/oxps**), and save it:

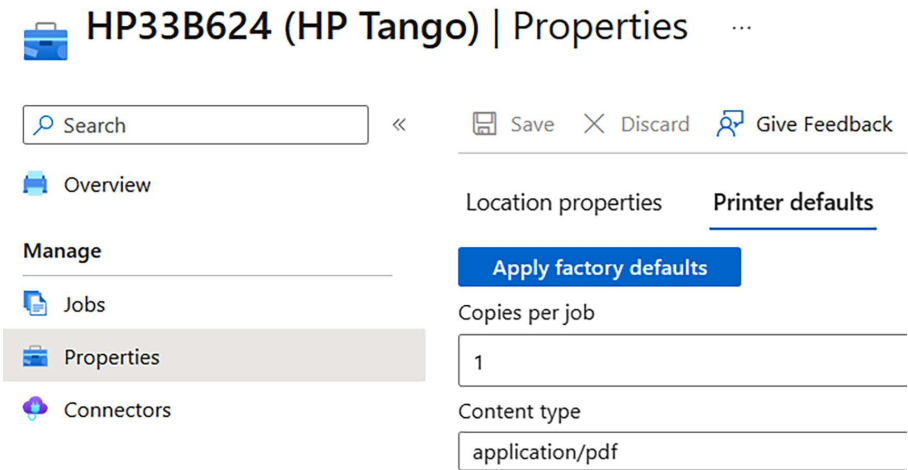


Figure 9.8: Setting printer content type

Together with inspecting/changing the content type for each printer, there is also another common parameter that needs to be set up in the Universal Print portal to be able to work smoothly with Dynamics 365 Business Central: document conversion.

Each printer accepts a specific document format instead of PDF, and since Dynamics 365 Business Central always sends a PDF file to be printed out, it is mandatory to enable this feature to parse and convert each document. To perform this task, simply go to the **Universal Print** home page, choose **Document conversion**, and enable it:

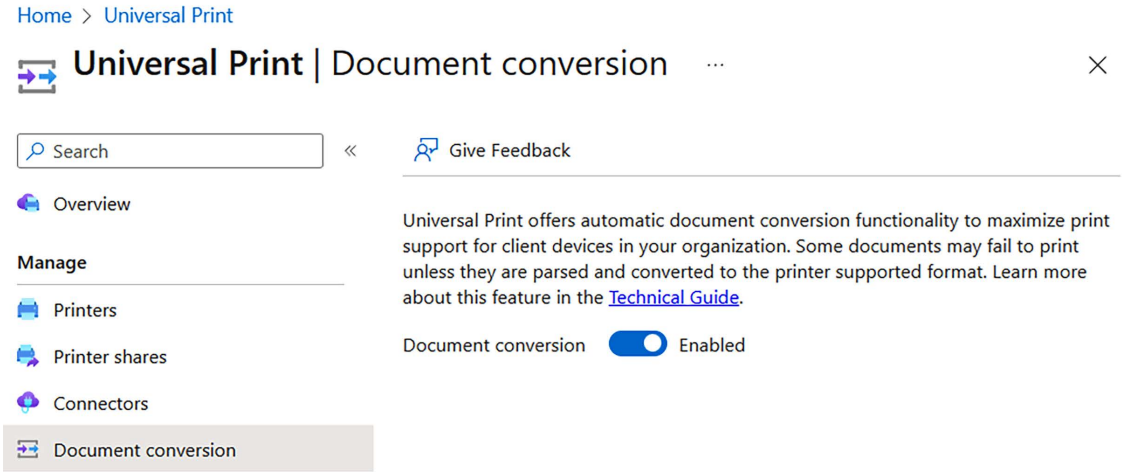


Figure 9.9: Universal Print document conversion

The last configuration task to accomplish is to share the printer with specific users or all in your organization. From the Universal Print home page, click on **Printer shares** and click **Add**. A panel will open letting you select a printer and add a share name, like the example provided below:

Share name \*

Select printer

HP33B624 (HP Tango)

Figure 9.10: Setting a printer share name

Once you have selected the printer and provided a name for the share, you can choose which users in your company this printer must be shared with, or toggle **Allow access to everyone in my organization** to share it with every user.

Selected printer:

---

HP33B624 (HP Tango)

---

☒ Allow access to everyone in my organization

Select member(s)

Figure 9.11: Setting access to a shared printer

To finalize the task and share the printer to be used by selected users, click **Share printer**. Once completed, a pop-up message like the one below should be displayed:

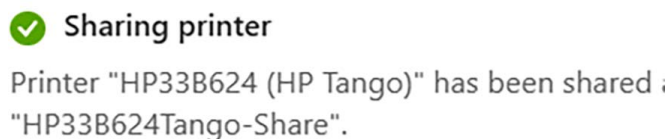


Figure 9.12: Sharing printer pop-up

Good! Everything is now perfectly set up in the online catalog. The last step is to import and configure the Universal Print printers in Dynamics 365 Business Central. To do so, there is a specific Microsoft extension: **Universal Print Integration**.

If you want to know more about further specific Universal Print features, please visit <https://learn.microsoft.com/en-us/universal-print/>

## Universal Print Integration extension

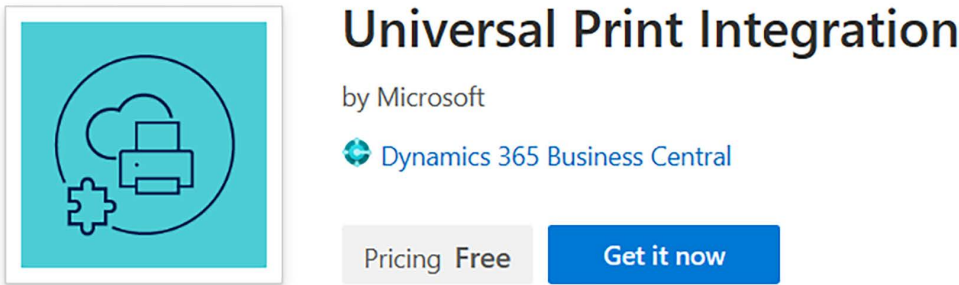


Figure 9.13: Universal Print Integration extension

The Universal Print Integration extension is a standard Microsoft AppSource app that is automatically deployed in every new environment, since October 2020. Its first version was deployed with Dynamics 365 Business Central 2020 Wave 2 (version 17.x). The code is open sourced on GitHub at <https://github.com/microsoft/ALAppExtensions/tree/main/Apps/W1/MicrosoftUniversalPrint>.

If you need to add minor changes or enhance it, feel free to create your own pull request and the product group will process it.

This app is super simple and is currently made of just 12 objects (2 tables, 4 pages, 1 page extension, 3 codeunits, and 2 enums) plus its permission sets. Its core processing code is in the 3 codeunits: setup, document ready, and graph helper. All in all, it is just a standard API connector extension that provides bridging from the Dynamics 365 Business Central environment PDF printouts to the Universal Print online catalog.

When deployed, you should be able to see the **Universal Print** action group on the **Printer Management** page in Dynamics 365 Business Central, as shown below:

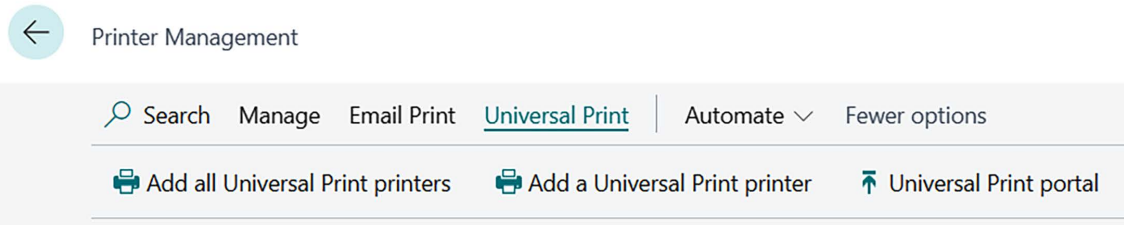


Figure 9.14: Universal Print action under Print Management

It is possible, then, to manually add a specific one or all the Universal Print printers (through a wizard) from the online catalog. Let us try this last option.

Since we have everything already configured in Azure, the wizard will not require any further parameters to be added and will just notify us of each action that it will take. Simply click **Next** until it finishes:

### Add Universal Print Printers

**That's it!**

Printers that are shared with you through Universal Print have been added to Business Central.

Number of printers added: 1.

Now you are ready to use Universal Print printers inside Business Central.

Figure 9.15: Walking through the printer installation wizard

In the end, it should result in a record on the **Printer Management** page. Drilling through the printer's name from that page, you should be able to see its properties:

# HP33B624TANGO-SHARE

NewAutomate Fewer options

NameHP33B624TANGO-SHARE

Print Share in Universal PrintHP33B624Tango-Share

DescriptionSends print jobs to the HP33B624TANGO-SHARE.

Paper SizeA4 paper (210 mm by 297 mm).

Paper Trayauto

Landscape

This feature utilizes Microsoft Universal Print. By continuing you are affirming that you understand that the data handling and compliance standards of Microsoft Universal Print may not be the same as those provided by Microsoft Dynamics 365 Business Central. Please consult the documentation for Universal Print to learn more.

Figure 9.16: Printer properties

You could then think of setting this printer as your own default printer in Dynamics 365 Business Central from the **Printer Management** page, by selecting the printer record and clicking on the **Set as my default printer** action. This will create a record in the **Printer Selection** table with the **User ID** and **Printer Name** fields populated.

When a cloud printer is assigned to a user, as shown before, every report request page will default to that printer and replace the **Handled by the browser** note:



*Figure 9.17: Printer action no longer handled by the browser*

Clicking the **Print** action from any request page will directly send the print output to the specified printer.

But what is really happening behind the scenes? Let us analyze it more deeply by exploring the Dynamics 365 Business Central report printing framework.

## Deep dive into the modern printing structure

If you think that printing a report is an easy matter that can be reduced to sending output to a printer, well, sorry, but that is not the case. Report printing, from a web application, is really a complex matter that involves a sequence of triggers. Its flow chart (basically its decision tree) has been published in detail by Microsoft here: **Report Triggers and Runtime Operations**, <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-report-triggers>.

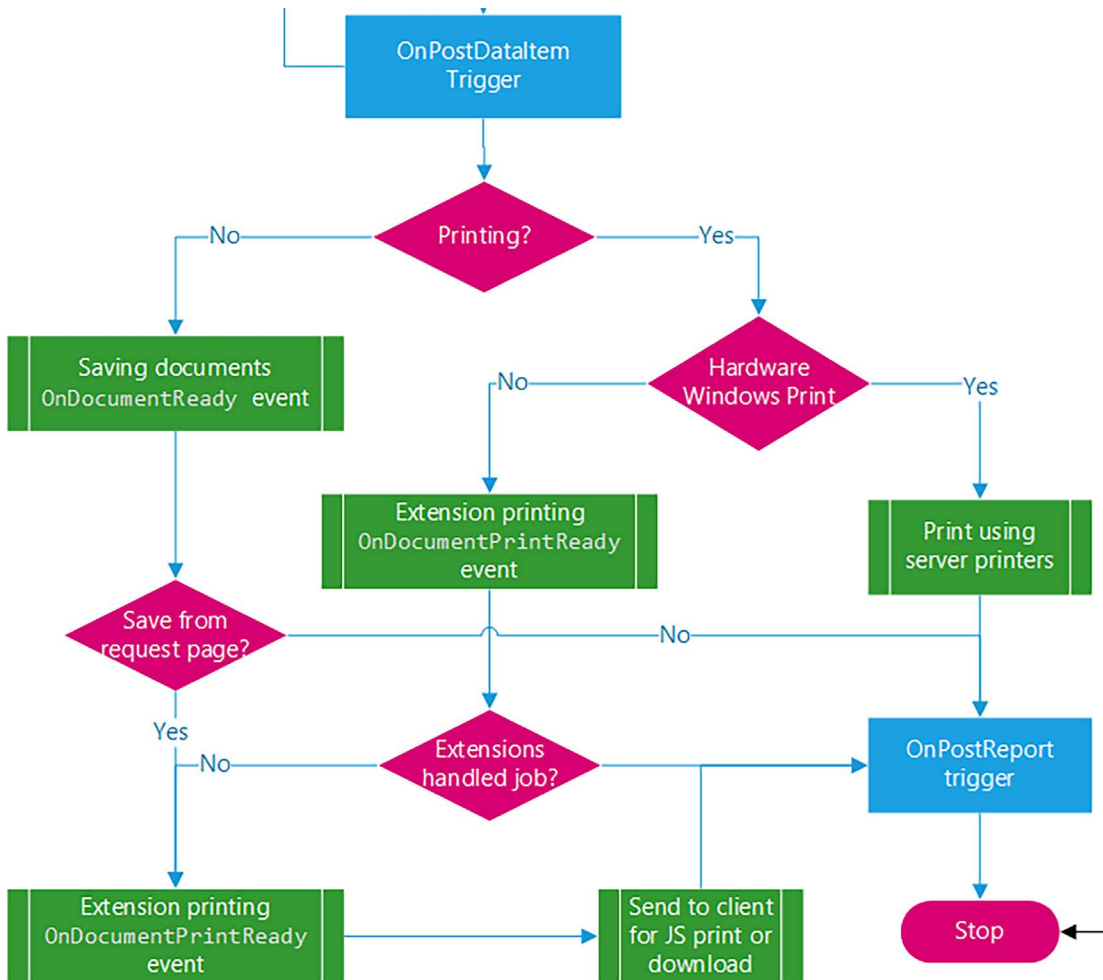


Figure 9.18: Report printing flow chart

Reading it carefully, you might notice that the main functions are *OnDocumentReady* and *OnDocumentPrintReady*. The source of these functions, along with being the source of the code and event publishers, is *codeunit 44 ReportManagement*, which further refers back to *codeunit 2000000005 Reporting Triggers*. This last object is important in the printing equation, and it is generated by the platform, while its symbol definition for references is contained in the *system.app* file. All its code content is made of business event subscribers handled directly by the platform itself.

To demonstrate the detailed system event sequence triggered when running a report, we have prepared a simple extension that subscribes to the relevant publishers and writes a sequence file (`EventSequence.txt`) that is stored in a log table together with the intermediate artifact files generated and the final output. In short, it is logging and keeping track of, in a time sequence, the various event publishers called during a specific process.

You can download the extension here: <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition>.

Once you have downloaded and deployed the app into your online sandbox, search for `PKT Report Triggers List` and select it. It should show the following:

PKT Report Triggers List: All ▾ | 🔍 Search | + New | 🗑 Delete | 🛠 Edit List | 📄 ShowLogs | Reports ▾

TextField ↑	CodeField	IntegerField	BigIntField	DecimalFieldN...	DecimailField...
Text1	CODE1	1	1000	1,56	2,778
Text2	CODE2	2	2000	3,11	5,555
Text3	CODE3	3	3000	4,67	8,333

Figure 9.19: PKT Report Triggers List extension

Within the extension, there is an install codeunit that creates a few records in a couple of tables that are simply used by the test *report 50700 PKT Demo Report Triggers* to generate some output. The demo report has been created with a multi-layout render section that supports layouts of the types RDLC, Word, Excel, and custom. The report output values are not relevant per se, since we are simply focusing on the trigger event sequence. Here’s the sequence of events in more detail:

1. Generic triggers containing body code from *codeunit 44 Report Management*, such as:
- `OnAfterDocumentReady`
  - `OnAfterIntermediateDocumentReady`
  - `OnAfterDocumentDownload`
  - `OnApplicationReportMergeStrategy`

The trigger that handled the body code will capture the generated report artifact and store it in a log table (*table 50703PKT Report Triggers Log Table*). The file name is generated from data in the report payload and saved to the log table, using the trigger name.

2. Log-only subscribers from *codeunit 44 Report Management*, such as:
- `OnAfterGetPaperTrayForReport`
  - `OnAfterGetPrinterName`
  - `OnAfterHasCustomLayout`
  - `OnAfterSubstituteReport`

3. Extension printer triggers from *codeunit 2000000005 Reporting Triggers*, such as:

- SetupPrinters
- OnDocumentPrintReady

Within the page list, in the **Reports** action group, you will find several actions:

Action	Layout type	Code snippet example
Report.SaveAs – Pdf	Word, RDLC, custom, Word as custom	<pre>SetSelection(ReportLayoutType::Word, ''); TriggerReport.SaveAs('', ReportFormat::Pdf, OutStr);</pre>
Report.Run – Implicit Print	Word, RDLC, custom, Word as custom	<pre>SetSelection(ReportLayoutType::Word, 'WordLayout'); TriggerReport.UseRequestPage := false; TriggerReport.Run();</pre>
Report.Print	Word, RDLC, custom	<pre>SetSelection(ReportLayoutType::Word, 'WordLayout'); TriggerReport.Print('');</pre>
Run Object	User selection	<pre>RunObject = report "PKT Demo Report Triggers";</pre>

Table 9.2: Actions in the Reports action group



The report layout is selected via AL code in almost all actions; if you want to run the test report with its default layout, then click on **Reset Layout Selection**.

Let us run one of them and inspect the results together:

1. From **PKT Report Triggers List**, click on the **Reports** action menu.
2. Choose **Report.Run - Implicit Print (RDLC)**.

The application will run the demo report and generate a message like the following, together with downloading a PDF file:

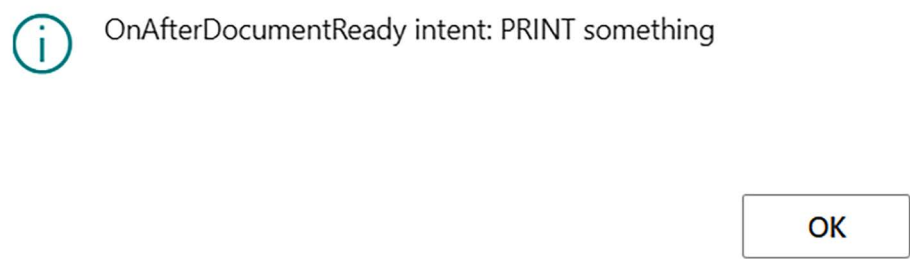


Figure 9.20: Application notification after running demo

- 3. Click **OK**. It will show the same message, but this time for the **OnAfterDocumentDownload** trigger. Click **OK** to also close this last message.
- 4. Click on the **Show Logs** action. A list page should open similar to the following, with a list of records logged:

View - PKT Report Triggers Logs

🔍

Search

+

New

📄

Edit List

🗑

Delete

⬇

Download File

🔗

☰

Entry No. ↑	Type	Output File Name	Output File	Is Text Encoded
→ 1	JsonFile	OnAfterDocumentReady.json		<input checked="" type="checkbox"/>
2	JsonFile	OnDocumentPrintReady.json		<input checked="" type="checkbox"/>
3	FileHandler-OnDocumen...	PKT Demo Report Triggers.pdf		<input type="checkbox"/>
4	JsonFile	OnAfterDocumentDownload.json		<input checked="" type="checkbox"/>
5	FileHandler-OnAfterDocu...	DownLoad - PKT Demo Report Triggers.pdf		<input type="checkbox"/>
6	EventSequenceFile	EventSequence.txt		<input checked="" type="checkbox"/>

Figure 9.21: PKT Report Triggers logs records

This is the list of intermediate artifacts generated during the report execution. By selecting any of these rows and clicking the **Download File** action, you will download the relevant artifact.

- 5. Select the row with the type **EventSequenceFile** and click on **Download File**.
- 6. Open the file just downloaded (EventSequence.txt). It should look like the following:

```
OnInitReport
OnInit
OnAfterHasCustomLayout
OnAfterGetPrinterName
SetupPrinters
OnAfterGetPaperTrayForReport
OnApplicationReportMergeStrategy
OnPreReport
ReportTriggers - OnPreDataItem
...
```

```

ReportTriggers - OnPostDataItem
OnAfterDocumentReady
OnDocumentPrintReady
OnAfterDocumentDownload
OnPostReport

```

What is, then, the scope of running this report? It is intended to determine the exact event sequence and what is provided by the platform within each event. Some of these events provide so-called intermediate printing artifacts where you could either manipulate their data and metadata structure or redirect them to a different destination.

For example, you could inspect the JSON object created in `OnDocumentPrintReady` and use this file to stream the document to a cloud printing service or a local service, through an API call. Below is a snippet of the JSON object:

```

    "filterviews": [],
    "version": 1,
    "objectname": "PKT Demo Report Triggers",
    "objectid": 50700,
    "documenttype": "application/pdf",
    "invokedby": "ffd62224-c3cc-4ad9-9eee-93aea1cf99b7",
    "invokeddatetime": "2024-01-23T03:10:49.247+00:00",
    "companyname": "CRONUS USA, Inc.",
    "printrname": "ReportTriggersPrinter",
    "duplex": true,
    "color": true,
    "defaultcopies": 7,
    "papertray": {
        "papersourcekind": 10,
        "paperkind": 24,
        "landscape": false,
        "units": 0,
        "height": 2200,
        "width": 1700
    },
    "intent": "Print",
    "layoutmodel": "Rdlc",
    "layoutname": "RDLCLayout",
    "layoutmimetype": "",
    "layoutapplicationid": "00000000-0000-0000-0000-000000000000",
    "reportrunid": "ac99bfce-088f-4ee9-b3f1-4b8d0591014a"
}

```

A similar approach has been used in the Microsoft standard Universal Print extension. So now, let us look at which of these useful events it is subscribed to and understand the backbone of its report printing execution flow.

Primarily, let us analyze the OnDocumentPrintReady subscriber from codeunit 2751 **Universal Print Document Ready** in the **Universal Print Integration** extension. See below:

```
[EventSubscriber(ObjectType::Codeunit, Codeunit::"Reporting Triggers",
'OnDocumentPrintReady', '', true, true)]
local procedure OnDocumentPrintReady(ObjectType: Option "Report","Page";
ObjectId: Integer; ObjectPayload: JsonObject; DocumentStream: InStream; var
Success: Boolean);
var
    UniversalPrinterSettings: Record "Universal Printer Settings";
    PrinterNameToken: JsonToken;
    PropertyBag: JsonToken;
    PrinterName: Text[250];
    FileName: Text;
    DocumentType: Text;
    FileNameWithExtension: Text;
begin
    // exit if handled already
    if Success then
        exit;
    // exit if not report
    if ObjectType <> ObjectType::Report then
        exit;
    // exit if not Universal Print printer
    if ObjectPayload.Get('printername', PrinterNameToken) then
        PrinterName := CopyStr(PrinterNameToken.AsValue().AsText(), 1,
MaxStrLen(PrinterName));
        if not UniversalPrinterSettings.Get(PrinterName) then
            exit;
        if ObjectPayload.Get('objectname', PropertyBag) then
            FileName := PropertyBag.AsValue().AsText();
        if ObjectPayload.Get('documenttype', PropertyBag) then
            DocumentType := PropertyBag.AsValue().AsText();

        FileNameWithExtension := GetFileNameWithExtension(FileName, DocumentType);
        Success := SendPrintJob(UniversalPrinterSettings, DocumentStream,
FileNameWithExtension, DocumentType);
end;
```

Taking a deep look at its code, this is using `ObjectPayload` to determine the Universal Print printer name and file name to be generated. After that, it invokes `SendPrintJob` to start the printing job. This procedure makes use of **codeunit 2752 Universal Print Graph Helper** to call Universal Print APIs. Let's look at this in more depth:

1. `UniversalPrintGraphHelper.CreatePrintJobRequest`: Send a request to create a print job and get back the print job ID and document ID.
2. `UniversalPrintGraphHelper.CreateUploadSessionRequest`: Create an asynchronous request to upload (stream) data, based on the previous job and document IDs, and get back an upload URL if successful.
3. `UniversalPrintGraphHelper.UploadDataRequest`: Upload the data.
4. `UniversalPrintGraphHelper.StartPrintJobRequest`: Once data is uploaded, the printing job can start through a specific request.

To learn more about cloud printing via the Graph API, you should visit *print resource type*, <https://learn.microsoft.com/en-us/graph/api/resources/print?view=graph-rest-1.0>.

It should be clear enough now how cloud printers work in principle and how to manipulate the report output before sending it to a modern printer through APIs.


While Universal Print is the official suggested way to integrate printer devices with the cloud world, there are still some gaps that need to be filled in, at least at the time we are authoring this book.

The first gap that should be filled by Microsoft is the price. The Microsoft Universal Print offering is currently more oriented to a high-volume enterprise market, and it might not be affordable for all the companies in the SMB segment. What we hope to have in the future is a sort of “attachment” license between Dynamics 365 Business Central and Microsoft Universal Print to have a better economic proposition.

The other gaps are purely technical, and most of them are underlined on the ideas site for *Universal Print: Universal Print Feature Requests*, <https://techcommunity.microsoft.com/t5/universal-print-feature-requests/idb-p/UniversalPrintFeatures/tab/most-kudoed>.

The one that is most valuable for Dynamics 365 Business Central is the missing onboarding for the biggest player in the label printer industry:

**83**  
Upvotes

  
[You upvoted!](#)

## Get label printer vendors onboard

Looking Into It

Submitted by [JSN00](#) on Mar 02 2021 10:09 AM [19 Comments \(0 New\)](#)

Label printing is super critical for warehouse applications. We need rugged devices from Zebra. Alternatively Dymo, Brother(who has joined). [... View more](#)

Figure 9.22: Label printers in Universal Print

This limitation has generated, as of today, so many support requests that it has also been published officially in the Microsoft documentation: *Support for label printers in Universal Print*, <https://learn.microsoft.com/en-us/universal-print/fundamentals/universal-print-label-printing>. What to do, then, if you need to directly send the output to a label printer that is not Universal Print-ready? You must look for alternatives.

## Alternatives to Microsoft Universal Print

### 1. Investigate the market for suitable AppSource extensions

The first suggestion for alternatives is do not reinvent the wheel. A few years ago, if you had the chance to search the AppSource marketplace for solutions enabling direct printing to local or network printers, you might have fumbled with one or two apps. If you perform the same search today, you will find more than 30 of them, using several printing services, such as **PrintNode** (<https://www.printnode.com/en>) or **Printix** (<https://printix.net/>), just to name a couple. If your label printer is, for example, from **Zebra** (<https://www.zebra.com/us/en.html>), you could search for Zebra and flag **Business Central** in the filter pane and you would find several apps that are ready to serve you. You could also use **CentralQ** (<https://www.centralq.ai/>) – and this is our personal choice – to try to find the extension that you need on your behalf.

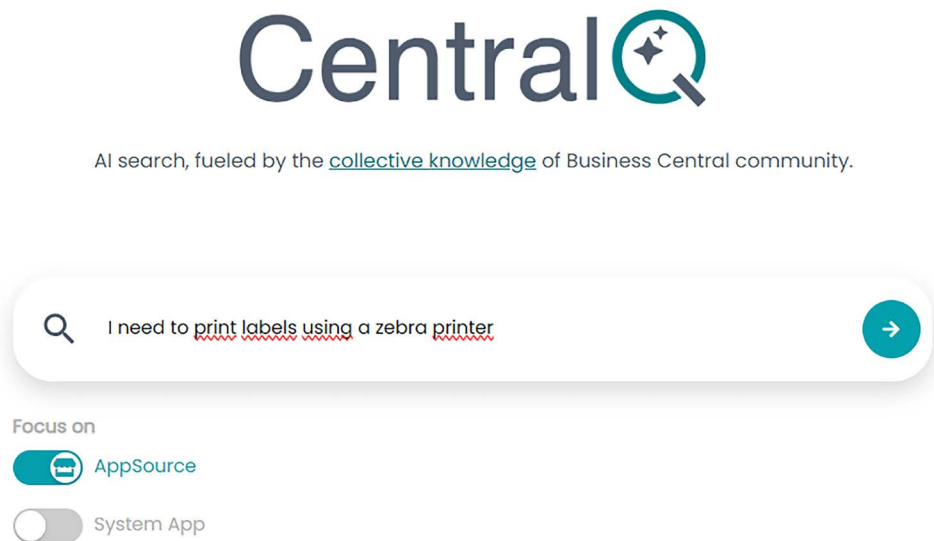


Figure 9.23: Using CentralQ to search for extensions

### 2. Subscribe to a cloud printing service and develop your integration extension

As alternatives to Universal Print, there are several other cloud printing services. Some of these have existed for several years and way before Microsoft thought about launching its own service. They are flexible and robust, and you will surely find one that is perfect for your printers and business strategy. Most of them have a trial period so that you can perform printing tests and implement the required API framework in AL.

### 3. Create your own specific printing solution, combining different technologies

Aside from using external cloud printing services, you might think of streaming an output file (typically in PDF format) into Azure Blob Storage via Azure Functions. Azure Blob Storage will then be checked by a local Windows service that will download the file locally and send it to a specific printer. You can read more about this solution here: *Dynamics 365 Business Central and direct printing*, <https://demiliani.com/2019/01/29/dynamics-365-business-central-and-direct-printing/>.

Alternatively, you could download the file locally from Azure Blob Storage using Power Automate and using Windows Task Scheduler to send the file to a printer through a PowerShell script.

Another way to mimic cloud printing is to create your own **BCAgent** (based on the Azure Relay service) with a printing plugin. To learn more about the BCAgent, you can check out the following sample: <https://github.com/microsoft/BCTech/tree/master/samples/BCAgent>.

In short, you can think of any way to automate the movement of the printing output locally, to have it printed in the way you like.

## Summary

In this chapter, we have learned the basics of cloud printers and introduced different printing services, with a special focus on Microsoft Universal Print. We have also had a deep dive into the complex printing framework implemented in the Dynamics 365 Business Central platform by using a demo app. Such a demo extension keeps track of every relevant trigger fired during report execution.

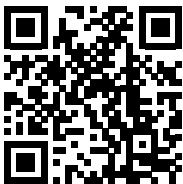
Now, you should feel equipped to discover the best printing strategy that would fit your or your customers' needs. You should also be able to decide whether to implement a custom solution or use an existing one from AppSource.

In the next chapter, we will master how to debug the application code with different tools.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>



# 10

## Debugging

The Dynamics 365 Business Central AL Language extension provides a debugger to help developers check, correct, and modify code so that custom extensions can build successfully, deploy smoothly, and act as expected.

Debugging can be done synchronously or asynchronously by collecting snapshots and analyzing them afterward. The purpose of the snapshot is to replay code execution for offline analysis. Together with debugging, it is also possible to analyze code performance by collecting AL profiler traces. Profiler traces can be collected using Visual Studio Code or directly from the client (in-client profiling).

There are also some useful external tools that can be used to analyze AL profiler traces.

This chapter will cover the following topics:

- Running the AL Language extension in debug mode
- Debugging in attach mode
- Snapshot debugging
- Performance profiling

### Running in debug mode

The basic concept behind debugging is to step through code execution, typically one statement at a time. A *breakpoint* is a mark that you can set on a statement to indicate where the debugger enters into the code execution. When the program flow hits the statement, the debugger kicks in and suspends execution (technically, it breaks) until instructed to continue. Without any breakpoints, the code would run just fine as long as the debugger is active.

The debugger will automatically stop the execution of the code only when it encounters an error, or if it has been instructed in the `launch.json` file to break on record writes.

A developer could also use the debugger to find potential logic errors since the debugger enables them to execute AL code syntax, one statement at a time, while inspecting the contents of variables at each runtime step. In this way, the developer can compare the runtime code execution against what they intended when they wrote the code at design time.

It is possible to create a debugger session from Visual Studio Code or, since Dynamics 365 Business Central 2023 Wave 2 (version 23), to create a debugger instance from scratch through the browser client. We will concentrate on running the debugger from Visual Studio Code rather than this last option. If you want to know more about attaching a debugger session directly from the client, we encourage you to read the following post when you have finished this paragraph: *Business Central 2023 wave 2 (BC23): Open Visual Studio Code (Debugging) from web client to investigate or troubleshoot extensions*: <https://yzhums.com/42334/>

You can run the debugger from Visual Studio Code in three ways:

- Click **Debug | Start Debugging**.
- Press the *F5* shortcut key.
- Go to the **DEBUG** view (*Ctrl + Shift + D*) and click the green right-facing arrow in the top bar. The top bar also shows the debugger session name specified in the `launch.json` file. It will open that file if you click the gear icon. Expanding the last icon on the right, it will let you show or hide different debugging view panes such as variables, breakpoints, or the debug console. The following screenshot depicts the debugger top bar:



Figure 10.1: Debugger shown on the top bar

All these actions will result in building your extension (equivalent to *Ctrl + Shift + B*), if this has not already been done, and then publishing the extension to the target online sandbox tenant as specified in the `launch.json` file.

It is also possible to run a debugger session without the need to build and publish the extension repeatedly. This helps reduce the debug cycle and increases development productivity. To try this, in Visual Studio Code, just hit *Ctrl + Shift + F5* or run the Command Palette (*Ctrl + Shift + P* or *F1*) and search for **AL: Debug without publishing**.

The `launch.json` file contains some elements that influence the debugging target, its behavior, and what to show or hide in the debugging pane. The following is a summary of the most relevant of them and what they do:

- **environmentType**: This is typically **Sandbox** but if the file entry is targeted for snapshot debugging, then **Production** or **OnPrem** can alternatively be specified as the environment type.
- **environmentName**: This parameter is where the developer specifies the name of the sandbox or production environment, in case of snapshot debugging, to connect to the debugger session.
- **tenant**: Specifies the **Azure Active Directory (AAD)** tenant name or its tenant ID (GUID) in which to create the debugger session.
- **breakOnError**: Specifies whether the debugger should stop when it hits an error. You could also specify to exclude errors coming from `TryFunction` procedures in order to focus only on real error breaks.

- `breakOnRecordWrite`: Specifies whether the debugger should stop on record changes (typically record create or update). Also, in this case, you could opt to refine this parameter by excluding record changes if a variable or table are marked as temporary.
- `enableSqlInformationDebugger`: Enables displays of SQL Server statements and related info in the debugging pane.
- `enableLongRunningSqlStatements`: Enables displays of a specific number of SQL statements that are marked as long-running based on a specific long-running SQL statement threshold.
- `useSystemSessionForDeployment`: Installs and upgrades code units. They will run using a system session, meaning that they cannot be hit by a debugger.
- `suppressMultipleSessionWarning`: This will not display a warning when the same debug configuration is started more than once.

Also, the `app.json` file contains several parameters that are vital for the debugger to work against specific extension code. The most important ones are contained in the `resourceExposurePolicy` array. This setting is the substitute for the old `showMyCode` and it contains a more granular distribution of the rules that determine both debugging capabilities and the handling of source code and symbols. It currently accepts four different values:

- `allowDebugging`: Enables or disables the debugging capability for the code contained in the extension.
- `allowDownloadingSource`: Determines if the source code can be downloaded for that specific app from the **Installed Extensions** page.
- `includeSourceInSymbolFile`: Specifically targeted to the `AL: Download Symbols` function. If set to false, the source code cannot be downloaded, hence IntelliSense features like `go to definition` cannot be used proficiently.
- `applyToDevExtension`: This is the latest resource exposure policy addition. Extensions deployed in sandboxes via Visual Studio Code (`F5` or `CTRL + F5`) behave as though all the aforementioned parameters are set to true, unless one is set to false. In this last case, the other three parameters are considered by the platform based on their values.

Be extremely careful in setting up these resource exposure policies since they not only let users debug code, but also enable them to download the source code for the extension using the standard Microsoft tools.

It is very important to stress the fact that a resource exposure policy could be a method of protecting your private **Intellectual Property (IP)**, but Microsoft has made it clear that there might be external tools that could extract your code even if you simply declare the enabling of a debugging feature. After all, this is an “*uroboro*”: if you allow debugging and prohibit downloading the source, then the code is shown in debugging mode, and everyone could perform a long and repeated drill to take note or copy parts of the code.

The source code accessibility vs. debugging discussion is so important that I would encourage you to read the official documentation about it twice: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-security-settings-and-ip-protection>.

The highest level of IP protection in SaaS could, then, be applied to AppSource apps only. It requires you to disable all resource exposure policies completely, including debugging, and use an override by creating an Azure Key Vault and storing a file that contains information (AAD tenant IDs) where you want to temporarily enable debugging capabilities (or whatever other resource policy change you want to apply for that specific tenant). Please note that this is valid only for AppSource extensions, as mentioned, and the procedure requires that you send an email to Microsoft to start the onboarding process. More information on the full procedure can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/setup-app-key-vault>.

Other parameters that could be useful to know or set up in relation to debugging are contained in the `settings.json` file. In the context of debugging, the most important are:

- `al.browser`: This lets you specify exactly which browser to start up out of the supported ones: Edge, Chrome, or Mozilla Firefox.
- `al.incognito`: In combination with the parameter above, this opens the browser in private/Incognito browsing mode. This is quite helpful if you have different customer environments and stored credentials.

## Debugging logs: verbose mode

In some real-world scenarios, it might be that, for some reason, the debugger will not start and reports an unhandled error message in the output window, or you might simply need to keep track of the debugger service process. In other words, you might need to “*debug the debugger*”. After all, the debugger is just another software artifact.

To gain more insight and verbose diagnostics, there is an undocumented feature that you need to enable by entering a specific parameter in the `settings.json` file:

```
"al.editorServicesLogLevel": "Verbose"
```

Once it is enabled, you need to restart Visual Studio Code to make the changes effective and let the debugger log in verbose mode.

This parameter will instruct the AL Language extension to use verbose logging for both the code editor (`EditorServices.log`) and the debugger (`DebuggerServices.log`) in the following directory:

```
C:\Users\<USER>\.vscode\extensions\ms-dynamics-smb.al-<runtime version>\bin\win32
```

The following is a snippet of the logged activity for the debugger service showing its processing:

```
02/17/2023 16:52:41 [/1] Editor Services Host v11.0.11.31731 starting (pid 45272)
02/17/2023 16:52:41 [/1] Editor Services Host started!
02/17/2023 16:53:09 [/20] Sending request to https://api.businesscentral.
dynamics.com/v2.0/SandboxGB/dev/metadata with request ID e218e0a4-c10f-44ce-a268-
bd9d7f6f28f1 and session ID 08a2c57a-07d8-4998-b47a-53ff56b7b5f7
02/17/2023 16:53:10 [/20] Sending request to https://
api.businesscentral.dynamics.com/v2.0/SandboxGB/dev/
apps?SchemaUpdateMode=synchronize&DependencyPublishingOption=default with request
ID 04e835c1-de1e-4074-863a-fa3fb4829f61 and session ID 08a2c57a-07d8-4998-b47a-
53ff56b7b5f7
```

```
02/17/2023 16:53:19 [/20] Establishing SignalR connection using ASP.NET Core SignalR
library
02/17/2023 16:55:16 [/1] Editor Services Host v11.0.11.31731 starting (pid
35468)...
02/17/2023 16:55:16 [/1] Editor Services Host started!
```

The following debugger output shows its processing in verbose mode:

```
setBreakpoints
02/20/2023 12:12:14 [/9] Parsing Report 50111 "Item Ledger Entry Analysis".
02/20/2023 12:12:14 [/9] Process:
    setBreakpoints
02/20/2023 12:12:14 [/9] Parsing Page 50102 "PKT Vendor Quality Card".
02/20/2023 12:12:14 [/7] Processing setbreakpoints took 0 seconds
02/20/2023 12:12:14 [/9] Process:
```

## Visual Studio Code debugger sections

The DEBUG view provides several sections and output windows to inspect, step by step, what's currently executing, the variable assignment status, and the code process flow. It is also possible to get some insight into code performance by gathering the longest-running database queries. Consider the following screenshot:

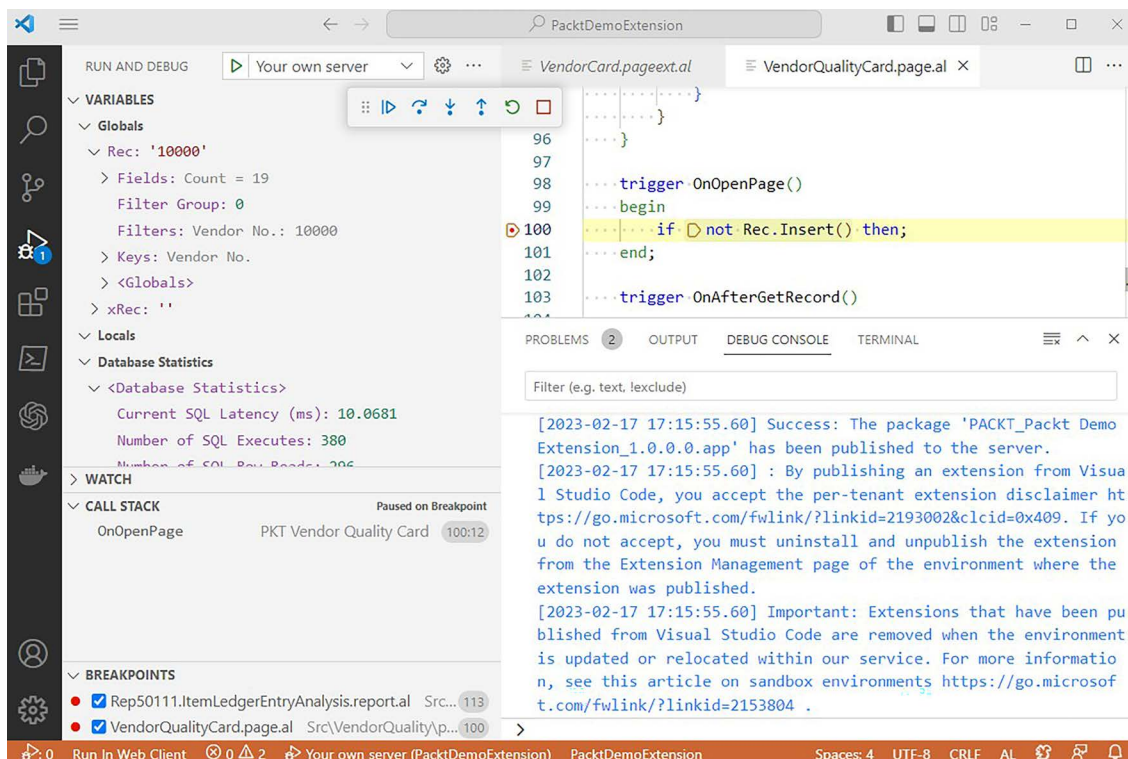


Figure 10.2: DEBUG view

The debugger is basically divided into four sections: the sidebar, the toolbar, and the editor and output windows. The editor window highlights where the code is currently stopped. This is typically marked in yellow. The debug console in the output window section shows debugging information.

## Debugger sidebar

The sidebar is enabled by default and is located on the left side of the debugger. It is possible to switch its position with the editor (right-click in one of the sections and select **Move sidebar right** – if you are familiar with Visual Studio, this might be your preferred choice), hide it (*Ctrl + B*), or even just hide some of the sections (right-click in one of the sections and uncheck the section(s) that you want to keep hidden).

The sidebar is divided into four sections that are used to provide information related to the current code flow. Let's go through them here.

## VARIABLES

The **VARIABLES** section provides an overview of global and local variable assignments:



Figure 10.3: VARIABLES section

In the **Database Statistics** tab, it is also possible to check performance counters related to code execution:

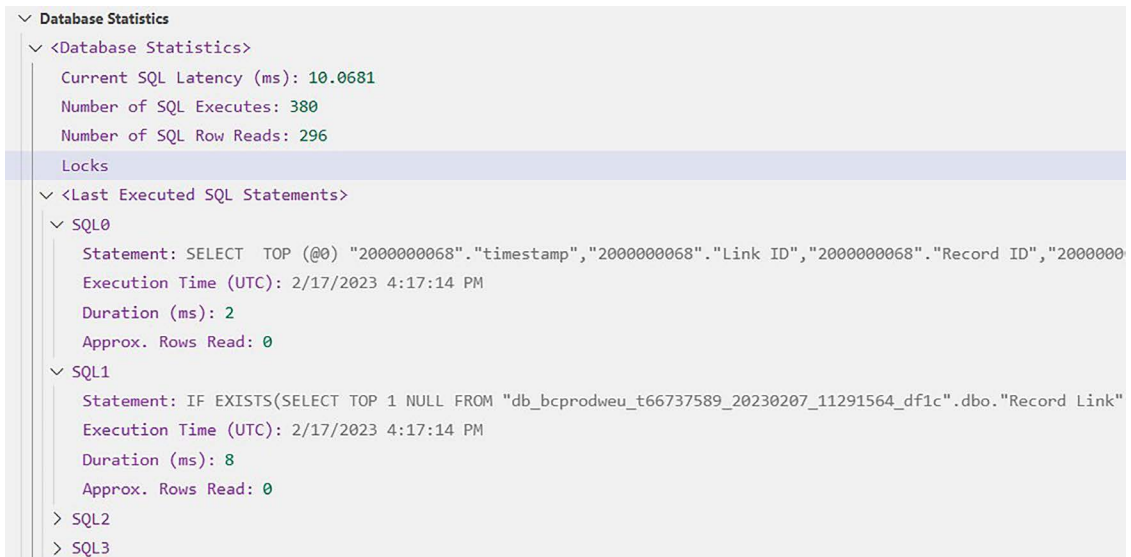


Figure 10.4: Database Statistics tab within VARIABLES

In particular, it is possible to measure the following:

- **Current SQL latency (ms):** When the debugger hits a breakpoint, the Dynamics 365 Business Central Server service will send a probing SQL statement to the Azure SQL database and keep track of how long it takes to receive an answer. This information is helpful if the sandbox node for the tenant has healthy latency or there are infrastructure issues.
- **Number of SQL executes:** The total number of SQL statements executed in the session since the debugger was started.
- **Number of SQL row reads:** The total number of database rows read since the debugger was started.
- **Locks:** This shows an overview of the locks held by the debugged session, if any.
- **Top-10 long-running queries:** Expanding the **Last Executed SQL Statements** section, you might observe up to 10 SQL Server statement entries (numbered from 0 to 9). The statements represent the 10 least performant queries, in terms of duration, that have been executed since the session began up to the first breakpoint hit. These are defined by the following elements:
  - **Statement:** The T-SQL statement executed. If a statement is quite big, as it typically is, only a small part is shown directly. You should hover over the statement to check the full statement.
  - **Execution time (UTC):** A timestamp defining when the SQL statement was executed.
  - **Duration (ms):** The duration of the total execution time of the SQL statement. It's useful to analyze this if there are some missing indexes when developing extensions.
  - **Approx. Rows Read:** Shows the approximate number of rows read by the SQL statement. It might be useful when looking for missing filters when developing extensions.

## Watch

The **Watch** section is used to monitor variables of particular interest while debugging. It is possible to right-click on the name of the variable that you want to watch from the **VARIABLES** window or in the code editor while debugging. This will display the value of the watched variable. In this window, you can also insert the names of the variables that you want to monitor into the watch list while debugging.

## Callstack

The callstack is nothing more than kind of a breadcrumb trail of how the code execution goes through objects. Variable values and expression evaluations are relative to the selected stack frame. This will report a cascade/stack of objects in descending execution order. This could be used to see how you got to the point in the code where you are right now. If you need to see variables in the context of the earlier calls, you could always set a breakpoint there and restart the debugger.

## Breakpoints

This shows a list of available breakpoints that could be enabled, disabled, or reapplied at will. Breakpoints can be toggled in the Visual Studio Code editor window by clicking in the left margin or by pressing *F9* on a selected line. Breakpoints that are displayed in the editor margin are shown as red-filled circles. Disabled breakpoints have a filled gray circle. Breakpoints that cannot be assigned to any code in the debugger session are shown with a gray hollow circle.

A handy topic on this point is conditional breakpoints, which are something that we did not have in C/SIDE, but we do in VS Code. Conditional breakpoints allow you to break only for a certain item or line number. This is very handy if you are debugging, for example, a sales line process for an order with 100 lines.

## Debugger toolbar

The toolbar contains commands that pause, stop, restart, or control the debugging process. The following screenshot shows a debugger toolbar:



Figure 10.5: Debugger toolbar

The possible actions are as follows:

- Continue (*F5*).
- Pause (*F6*).
- Stop (*Shift + F5*) depicted by the red square: The debugger toolbar commands allow you to continue (*F5*) the process until it comes to an end. In this way, developers can continue with their iterative process and start the operation again without running a new web client session in debug mode. The process could also be paused (*F6*, the debugger session will still be alive); restarted (*Shift + F11*, which will create a new debugger session); or definitively stopped (*Shift + F5*, which closes the debugger session).

- Step Over (*F10*): All statements are executed one at a time. If this command is used, when a function call is reached, the function is executed without the debugger stepping through the function instructions. If there is a breakpoint in one of the functions that it has been instructed to step over, the debugger will break at that breakpoint in any case.
- Step Into (*F11*): All statements are executed one at a time. If this command is used, when a function call is reached, the debugger will step through all the function's instructions.
- Step Out (*Shift + F11*): This continues to execute the remainder of the current function and will go back to where it was called from (back one step through the callstack). This is most often used when you hit *F11* when you really wanted to step over.

## Debugging in attach mode

Within the AL debugger, it is possible not only to launch it and create a new session but also to attach the debugger to the next session spun up by the application, or to an existing session. This is mostly used to debug web service calls (API, OData, or REST) within SaaS sandboxes but also install and upgrade codeunits within on-premises scenarios. Debugging in attach mode could also be done and is very useful in **service-to-service (S2S)** integration scenarios by specifying the session or user ID to attach to.

This capability currently has some limitations, and the following table explains its supportability scenario:

Deployment type	Web client	Web service	Background session
On-premises	Supported	Supported	Supported
Online sandbox	Not supported	Supported	Not supported

Table 10.1: Supportability comparison

To enable the attach functionality, it is mandatory to add a new configuration parameter set to the `launch.json` file with a different request type. You can easily do this by adding a comma (,) to the last array element and pressing *Ctrl + Space* as shown below:

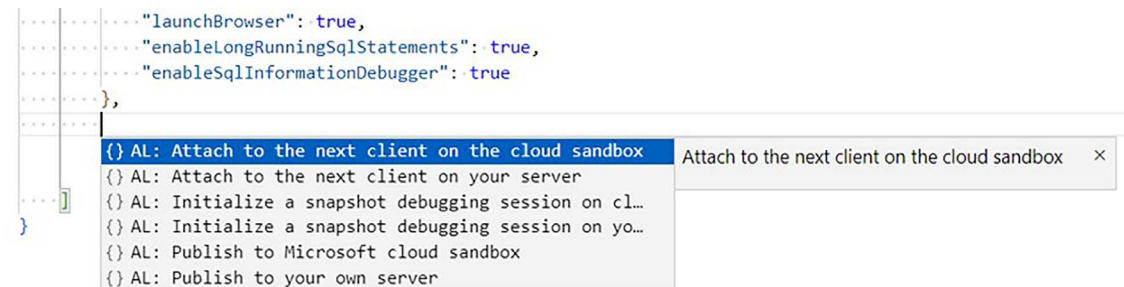


Figure 10.6: Enabling attach functionality

The key parameters that need to be specified are as follows:

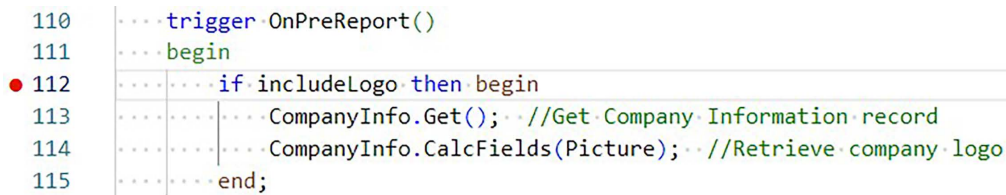
- "request": "attach": In a typical debugging scenario, this parameter is set to the default value: launch.
- "breakOnNext": "WebServiceClient": With online sandboxes, this is the only option allowed while with on-premises it is also possible to attach the debugger to "WebClient" or "Background" sessions.
- "sessionId": If specified, it will attach to the process specified by this session ID.
- "userId": If specified, it will attach to the next session spin-up by that specific user.

## Non-debuggable items

Typically, developers would like to have a full debugging experience on every line of code in an extension. However, there are some circumstances where a specific variable or function should not show its current value. These circumstances are typically related to variables that store private information, or functions that return private values (such as user passwords or license checks).

When developing extensions, there is a special attribute that can be used with functions and/or variables that stops them from being processed (the debugger cannot step into them) or visible (variables and/or function output values are not shown) within the debugger. Writing the `[NonDebuggable]` attribute decorator before the declaration of a function or a variable would mean that they are not inspectable, and no breakpoints could be set against them.

In the `Item Ledger Entry Analysis` report created in Chapter 8, *Report Development*, add a breakpoint in the `OnPreReport` trigger in the very first statement, `if includeLogo then begin`, as follows:



```

110  ... trigger OnPreReport()
111  ... begin
112  ...     if includeLogo then begin
113  ...         CompanyInfo.Get(); //Get Company Information record
114  ...         CompanyInfo.CalcFields(Picture); //Retrieve company logo
115  ...     end;

```

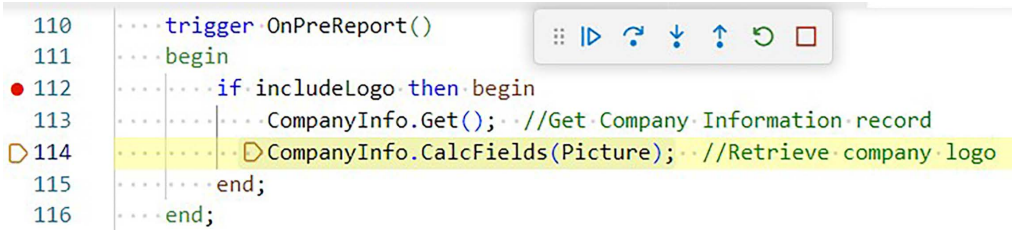
Figure 10.7: Breakpoint on line 112

And after that, just publish the extension (F5).

When the client loads, search for and run the `Item Ledger Entry Analysis` report, then choose to include the logo in the request page and click **Preview**.

The debugger will stop precisely on the `OnPreReport` breakpoint that was just added.

Now, press `F11` twice to move the code execution down to run the `Get` statement in the `Company Information` table:



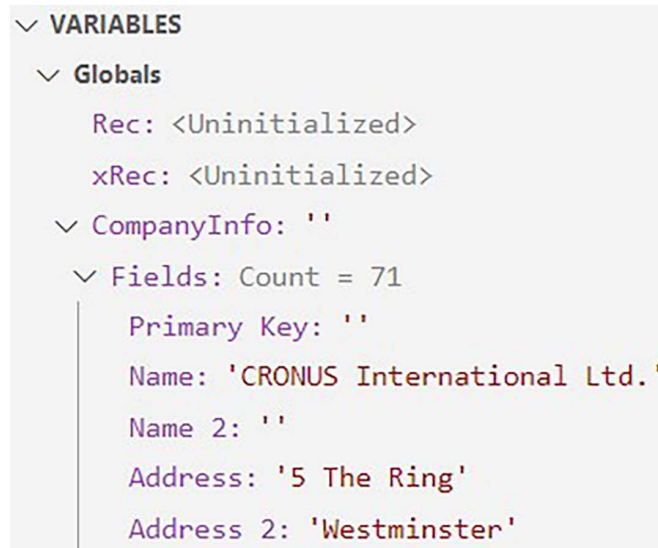
```

110 ... trigger OnPreReport()
111 ... begin
112 ... if includeLogo then begin
113 ...     CompanyInfo.Get(); //Get Company Information record
114 ...     CompanyInfo.CalcFields(Picture); //Retrieve company logo
115 ... end;
116 ... end;

```

Figure 10.8: Running the Get statement

If you expand the **VARIABLES** section in the debugger activity pane, you might notice that the company information (named `CompanyInfo`) record can be expanded, and you can see all its values:



```

▼ VARIABLES
  ▼ Globals
    Rec: <Uninitialized>
    xRec: <Uninitialized>
  ▼ CompanyInfo: ''
    ▼ Fields: Count = 71
      Primary Key: ''
      Name: 'CRONUS International Ltd.'
      Name 2: ''
      Address: '5 The Ring'
      Address 2: 'Westminster'

```

Figure 10.9: Company information in the VARIABLES section

Stop the debugger (*Shift + F5*) and add the `[NonDebuggable]` attribute before the `Company Information` global variable, as follows:

```

var
    [NonDebuggable]
    CompanyInfo: Record "Company Information";

```

Build and publish the extension (*F5*) again.

When the client loads, search for the `Item Ledger Entry Analysis` report and perform the same action as before: the debugger will stop once again in the same place.

Press *F11* twice to move the code execution down to run the `Get` statement in the `Company Information` table and retrieve the record data. Now, if you expand the **VARIABLES** section, you might notice that the `CompanyInfo` record is not even shown:



Figure 10.10: Hiding company information

Hovering the mouse over any `CompanyInfo` statement in the code editor will result in an `<Out of Scope>` value, due to the presence of the `[NonDebuggable]` attribute decorator for that specific variable.

Now that we have mastered the different synchronous debugging styles, let us have a look at how we could collect a debugger trace and analyze it offline asynchronously using the snapshot debugger.

## Snapshot debugging

As of the time of writing, none of the aforementioned debugging actions can be performed against an online production environment for security reasons. The only way to debug a production environment in the cloud is asynchronously by collecting a snapshot of the server-side code execution and analyzing it offline in Visual Studio Code. It is possible to collect any number of snapshots and apply specific breakpoints to them, called *snappoints*, that are not meant to stop the execution of the code at runtime but instruct the server to log and collect the call stack and state of the variables at the *snappoint* (or when the application goes into error).

To be able to create and/or collect snapshot debugger files, the user must be part of the **D365 SNAPSHOT DEBUG** standard permission set. This will provide the needed permission to the relevant objects involved in the snapshot debugging process, such as the `Published Application` system table.

We also need to declare a specific launch configuration, which is done in the `launch.json` file by pressing *Ctrl + Space* and selecting **AL: Initialize a snapshot debugging session on cloud** (or on your own server, if you are debugging on-premises). This will create a configuration with the following relevant parameters:

- `"name"`: This will be displayed when running the initialization.
- `"type"`: `"al"`.
- `"request"`: `"snapshotInitialize"`.
- `"tenant"`, `"environmentType"`, `"environmentName"`: This defines what to connect to. It could be either a sandbox or a production environment.

- "breakOnNext": This is one of the most important parameters, with which it is possible to collect snapshots from WebClient, WebServiceClient, and Background sessions.
- "executionContext": This instructs the server to initialize and collect a snapshot Debug trace or a performance Profile, or both (DebugAndProfile).
- "snapshotVerbosity": Depending on what you would like to inspect, you could decide to take a Full snapshot that includes the entire stack trace, or simply collect the stack trace and variable only when any SnapPoint is hit.
- "sessionId", "userId": As with classic debugging, if specified, it will try to connect to the specified session or user ID.

There are also a couple of setup parameters that drive the behavior of the AL: Initialize a snapshot debugging session on cloud option and are part of the `settings.json` file:

- "al.snapshotOutputPath": The default output path is set to the `.snapshots` directory in the current extension folder. This value determines where the snapshot files should be downloaded.
- "al.snapshotDebuggerLinesHitDecoration": {"gutterColor": "<HexValue>"}: It is possible to change the highlight color of the line decorator used to mark the line where the snappoint stops.

Finally, the following are some operational limits and a few things to always remember when performing snapshot debugging:

- Variables are populated only at snappoints.
- Symbols must match the ones used locally in Visual Studio Code.
- An initialized snapshot debugger session will stop after 30 minutes if there is no next session to attach to within this time frame.
- A snapshot debugger session can be recorded for a max of 10 minutes.

With all of this info, we can start using the snapshot debugger proficiently against a production environment deployment in a super simple example. Let's say that you would like to know if the **Item Ledger Entry Analysis** report has been run, with the option to print the the report with the company logo:

1. In Visual Studio Code, add a snappoint (*F9*) in Report 50111 Item Ledger Entry Analysis, in the OnPreReport trigger:

```

110 | ... trigger OnPreReport()
111 | ... begin
112 | ... if includeLogo then begin
113 | ...     CompanyInfo.Get(); //Get Company Information record
114 | ...     CompanyInfo.CalcFields(Picture); //Retrieve company logo

```

Figure 10.11: Adding a snappoint on line 112

2. Deploy **Packt Demo Extension** in a staging production environment through the **Installed Extensions** page using the **Upload Extension** action.

3. Log out and log in again. Once logged back in, click on the question mark ? at the top right and select **Help & Support**. Scroll down the page and take note of the **Session ID (server)**. See the following example:

Session ID (server): 25519

### Additional logging

Figure 10.12: Session ID in Help & Support

4. Back in Visual Studio Code, locate the `launch.json` file and add another JSON element in the launch array like the one shown below. Please note that `sessionId` should reflect the current server-side session ID retrieved in the previous step:

```
{
  "name": "snapshotInitialize: Microsoft production cloud",
  "type": "al",
  "request": "snapshotInitialize",
  "environmentType": "Production",
  "environmentName": "Production",
  "breakOnNext": "WebClient",
  "executionContext": "DebugAndProfile",
  "snapshotVerbosity": "Full",
  "sessionId": 25519
}
```

5. Press *F1* and select **AL: Initialize snapshot debugging** or simply press *F7*. Snapshot debugging will initialize the collection of a trace on the server side against the session ID specified in the `launch.json` file. In the Visual Studio Code output window, there should be verbose logging like the following:

```
[2023-03-13 10:26:41.54] Requesting metadata for a snapshot debugging
session.
[2023-03-13 10:26:41.55] Authenticating...
[2023-03-13 10:26:42.21] Authenticated.
[2023-03-13 10:26:42.21] Authenticated as user 'xxx@yyy.onmicrosoft.com'.
Please note that these credentials are cached. Clear the credentials
cache to authenticate as another user.
[2023-03-13 10:26:42.21] Targeting Dynamics 365 Business Central.
[2023-03-13 10:26:42.22] Sending request to https://api.businesscentral.
dynamics.com/v2.0/Production/snapshotdebugger/snapshotendpointmetadata
[2023-03-13 10:26:43.23] Initializing a snapshot debugging request on
debugging context : ba645810-fcb1-45ef-b0c7-add4c9a8344d
[2023-03-13 10:26:43.23] Sending request to https://api.
businesscentral.dynamics.com/v2.0/Production/snapshotdebugger/
```

```
attach?debuggingcontext=ba645810-fcb1-45ef-b0c7-
add4c9a8344d&sessionId=25519
```

```
[2023-03-13 10:26:48.34] The snapshot debugger initialize request for the
debugging context 'ba645810-fcb1-45ef-b0c7-add4c9a8344d' has succeeded.
Check the status of an initialized snapshot by using the 'Show all
snapshots' command, or by clicking the snapshot debugger icon in the left
corner of the toolbar.
```

6. By pressing *F1* and selecting **AL: Show all snapshots** or simply pressing *Shift + F7*, you can check the status of all the snapshots stored in your repository specified by the `al.snapshotOutputPath` parameter in `settings.json`:

The snapshot debug sessions

ba645810-fcb1-45ef-b0c7-add4c9a8344d

status:Started tenant:default user:The first user with a session on the tenant kind:UserSession

Figure 10.13: Checking snapshot status

The same can be achieved by clicking the snapshot debugger icon at the bottom left in Visual Studio Code. The following screenshot shows that, as an example, there are six snapshot traces in the output folder:



Figure 10.14: Checking snapshot traces

7. Now that the snapshot has been initialized, go back to the web client, search for the **Item Ledger Entry Analysis** report, and run it.
8. On the request page, check the option to include the logo and click **Print**.
9. Go back to Visual Studio Code, then press *F1* and select **AL: Finish snapshot debugging on the server**, or just press *Alt + F7*. This action will stop collecting the server-side snapshot trace and, once stopped, it will be downloaded automatically to the repository specified by the `al.snapshotOutputPath` parameter in `settings.json`. Once again, you might notice some verbose logs in the debugger output window:

```
[2023-03-13 10:49:27.79] Finishing a snapshot debugger attach request on
debugging context : 'ba645810-fcb1-45ef-b0c7-add4c9a8344d'
[2023-03-13 10:49:27.79] Sending request to https://api.
businesscentral.dynamics.com/v2.0/Production/snapshotdebugger/
finish?debuggingcontext=ba645810-fcb1-45ef-b0c7-add4c9a8344d&applicationg
atewayaffinity=k50X7gxx
[2023-03-13 10:49:28.85] The snapshot debugger attach session
has finished. Snapshot successfully saved as 'c:\APP\Mastering2\
PacktDemoExtension\.snapshotOutput\ba645810-fcb1-45ef-b0c7-add4c9a8344d.
zip'.
```

As the snapshot can contain customer privacy data, you should handle it according to privacy compliance and remember to delete it when you are done using it.

Use the 'Show all snapshots' command, or by clicking the snapshot debugger icon in the left corner of the toolbar to view the downloaded snapshots ready for debugging.

10. Now, your snapshot debugging trace should be downloaded into your local repository in ZIP format, and you can start snapshot debugging whenever you like and as many times as you like by pressing *Shift + F7* (AL: **Show all snapshots**) and selecting the trace that has been just saved.
11. The debugger should stop precisely on the first error or, in this case, on all snappoints that have been added. When stopping on snappoints, you might notice that local and global variables are shown and populated in the debugger views. The `includeLogo` variable will be set to `True` in our example:

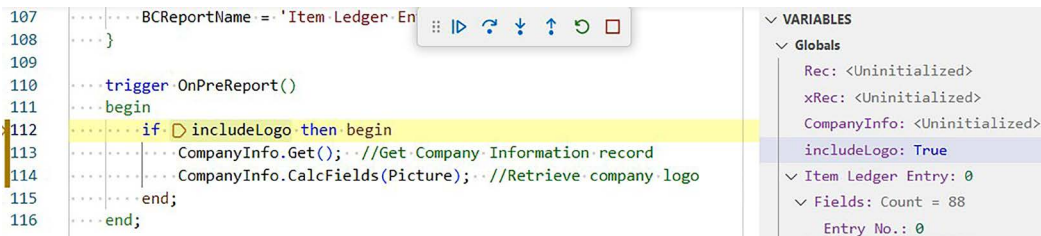


Figure 10.15: Setting the `includeLogo` variable

12. Now, try pressing *F10* to step over the next statement and you will notice that all variable statuses are gone, and a message is shown as follows:

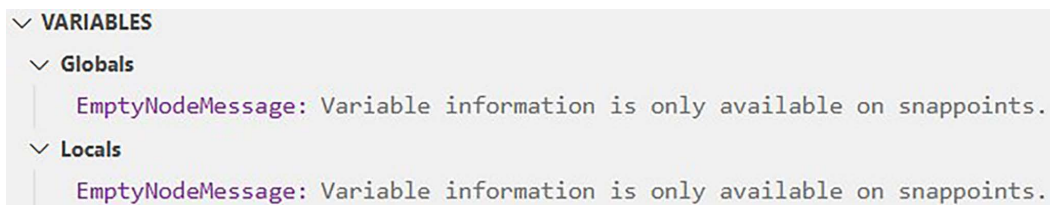


Figure 10.16: Variable status not collected

This happens because the next statement has not been declared under a snappoint, hence it will not collect the variable status at that specific moment in time. If you want to see the variable values at any point in the code, you need to add snappoints at design time to collect variable information.

Due to its asynchronous and fragmented nature, snapshot debugging is used typically in production environments. When it is not possible to replicate the same problem in a sandbox environment, using the classic debugger is preferred. Snapshots are also helpful when involving Microsoft in development or application performance support requests.

Now that we have learned how to debug in a couple of different flavors – synchronous (standard) and asynchronous (snapshot), in the next section, we will dig deeper into code performance using the AL profiler.

## Performance profiling

With or without a snapshot debugger log, the Dynamics 365 Business Central service can be instructed to collect an AL code performance profiler trace.

As with snapshot debugging, profiling is tightly related to the request parameter, which must be set to `snapshotinitialize`, and overall, by the `executionContext` property specified in the `launch.json` file. This last one can be configured to collect only an AL Profiler trace or both snapshot and profiler traces (`DebugAndProfiler`).

Another important parameter that influences the frequency of value collection within a given time is the "profilingType" parameter specified in the `launch.json` file. Profiling can be of two different types:

- **Instrumentation**, where any AL procedure is wrapped in telemetry instrumentalational code that captures time measures.
- **Sampling**, where measures are collected only within very specific and constant sampling intervals through a separate collection process.

This sampling interval constant is set by default to 100 milliseconds and can only be changed to 50 or 150 milliseconds. It is always recommended to opt for the instrumentation profiling type, unless you have clear lengthy processes for which sampling could be more useful to avoid collecting useless or unnecessary data.

An AL profiler trace is generated from the snapshot. At this stage, we will use the one that we created in the previous section:

1. Run the Command Palette (*Ctrl + Shift + P* or *F1*), then search for and select the **AL : Generate Profile** file.
2. Select the snapshot debug session file collected (in our last example, it was `ba645810-fcb1-45ef-b0c7-add4c9a8344d.zip`, but yours will probably have a different GUID).

An AL profile file should then be created and opened by the AL runtime. This will be placed together with the snapshot debug session file, with the same name but a different extension of `.alcuprofile`.



Figure 10.17: AL profile file

The AL profile file should look like the following:

Filter functions or files, or start a query()

Figure 10.18: Viewing an AL profile file

The measures collected in an AL profiler trace are:

- **Self Time (ms):** This is the time spent by the AL statement execution on a specific procedure, without considering the time spent on any other call outside that specific procedure. This is also expressed as a percentage of the total time spent by AL statements on the same procedure.
- **Total Time (ms):** This is the Self-Time value plus the amount of time spent on any other function call performed within that specific procedure. The % value that you find to the right of **Total Time** reflects the proportion of time spent by other function calls on the specific procedure.
- **Hit count:** The number of times a specific procedure was executed during the profiler trace collection.

Consider the following AL profiler snippet:

Self Time in ms		Total Time in ms		Hit count	Function Name
39.05	50%	76.99	50%	1	▼ OnOpenPage
37.64	87%	43.05	13%	1	▼ SetPackageTrackingVisibility
5.41	12%	42.81	88%	1	> IsEnabled
0.30	5%	5.73	95%	1	> SetDimVisibility
10.51	100%	10.51	0%	150	GetCurrencyCode
3.55	100%	3.55	0%	49	GetCaption

Figure 10.19: AL profiler snippet

The AL statements executed within the OnOpenPage trigger represent 50% of the time spent on that trigger. The other 50% of the time was spent calling other procedures from and within the OnOpenPage trigger: SetPackageTrackingVisibility and SetDimVisibility. It is possible to drill down to the entire call stack and inspect the metrics of every single child node call.

The GetCurrencyCode and GetCaption procedures do not have any other function calls, hence Self Time displays as 100% genuine AL statements within these procedures and there is no option to drill down further. It is worth noticing that these two procedures were called 150 and 49 times, respectively, during the AL profiler trace collection.

Both time and hit count values can be a good indicator of where a performance problem lies, but equally they can simply be used as tips to improve the code performance by refactoring the code within a lengthy procedure or reducing the number of calls to it.

In this specific case, even though it is called 150 times, it only used for 10 milliseconds, which is very quick and doesn't cause any performance issue. Quite often, such procedures escape at the start, so a high number of calls does not automatically mean that it is a bottleneck. All in all, it is important to look at code efficiencies but also to look at all elements of a trace like this, so you do not have to spend a lot of time removing a bunch of function calls if these are not causing real performance issues.

When an AL profiler trace file is opened, it always defaults to a **top-down** view where you can drill down through child functions starting from their parents. In the top-right corner of the AL profiler graph, there is an icon button to switch from the top-down to a **bottom-up** view to sort lines in a reverse call-stack mode.

The function name is decorated with a small square shown in different colors. The given color depends on the extension type these functions belong to. Colors can be changed at will by setting the "al.profilerColors" parameter in the settings.json file. The following snippet shows this parameter's default values:

```
"al.profilerColors": {  
  "apps": [],  
  "baseApplication": "magenta",  
  "extension": "yellow",  
  "system": "blue",  
  "systemApplication": "green"},
```



The "apps" is a JSON array element and can be configured by adding several extension names and color pairings. In this way, it would be easy to focus on specific values and find out to which extension, or extensions, the problem belongs.

But what does an AL profiler trace file really look like? In Visual Studio Code, right-click the (or any) .alcuprofile file and select **Reveal in File Explorer** (*Shift + Alt + R*). Copy the file and change its extension to .json. Double-click on the file to let Visual Studio Code open it. As you can see, this is just JSON that defines single or nested nodes. Let's look through an example node and explain the parameters as we go:

```

    "id": 1,
    "callFrame": {
      "functionName": "OnOpenPage",
      "scriptId": "Page_38",
      "url": "al-preview://allang/PacktDemoExtension/Page/38/Page_38.dal",
      "lineNumber": 530,
      "columnNumber": 8
    },
    "hitCount": 1,
    "children": [
      2,
      7
    ],

```

Let's look at the key parameters we've highlighted here:

- **"callFrame"**: This represents the exact position of the function in the source code. The **"url"** parameter enables the drill-through-symbols capability. This is shown in the UI as a hyperlink on the right side of every line. Clicking the hyperlink will load the relevant source code file. The **"lineNumber"** and **"columnNumber"** parameters define exactly where to position the cursor.
- **"hitCount"**: The number of times the function is called within the trace.
- **"children"**: This defines the tree structures and the number of nodes, if any. In the previous example, you might notice it had the values [2,7]. This means that there are two separate nodes (SetPackageTrackingVisibility and SetDimVisibility) with a total of seven function subcalls in the tree structures:

Function Name	
OnOpenPage	Base Application::Page::"Item Ledger Entries" Page_38.dal:530
SetPackageTrackingVisibility	Base Application::Page::"Item Ledger Entries" Page_38.dal:654
Enabled	Base Application::CodeUnit::"Package Management" CodeUnit_6516.dal:18
Enabled	System Application::CodeUnit::"Feature Management Facade" CodeUnit_2611.dal:22
Enabled	System Application::CodeUnit::"Feature Management Impl." CodeUnit_2610.dal:316
InitializeFeatureDataUpdateStatus	System Application::CodeUnit::"Feature Management Impl." CodeUnit_2610.dal:80
SetDimVisibility	Base Application::Page::"Item Ledger Entries" Page_38.dal:559
UseShortcutDims	Base Application::CodeUnit::DimensionManagement CodeUnit_408.dal:2584

Figure 10.20: Example tree structure with two nodes and seven functions

- Now, back to the node example. This continues straight from the children parameter:

```

    "positionTicks": [
      {
        "line": 548,
        "column": 8,
        "ticks": 1,
        "executionTime": 43050
      },
      {
        "line": 549,
        "column": 8,
        "ticks": 1,
        "executionTime": 5726
      }
    ],

```

Let's look a bit more closely at the highlighted parameter:

- "positionTicks": This represents the position in the source code of every node, if any exist, and their execution times. Returning to the previous example, there are two nodes defined by the "children" parameter. These reflect the declarations in the source code file:

```

541 | ... trigger OnOpenPage()
542 | ... begin
543 | | ... OnBeforeOpenPage();
544 |
545 | | ... if (Rec.GetFilters() <> '') and not Rec.Find() then
546 | | | ... if Rec.FindFirst() then;
547 | |
548 | | ... SetPackageTrackingVisibility();
549 | | ... SetDimVisibility();
550 | ... end;

```

Figure 10.21: Nodes shown in the source code

Back again to the node example. The following code completes the node and follows directly from the positionticks parameter:

```

    "declaringApplication": {
      "appName": "Base Application",
      "appPublisher": "Microsoft",
      "appVersion": "23.0.12034.12676"
    },
    "applicationDefinition": {
      "objectType": "Page",
      "objectName": "\"Item Ledger Entries\"",

```

```

        "objectId": 38
      },
      "isIncompleteMeasurement": false,
      "isBuiltinCodeUnitCall": false,
      "frameIdentifier": 269013750,
      "startTime": 63814297731661786,
      "endTime": 63814297731738774
    },
  ],

```

The highlighted parameters are explained below:

- "declaringApplication": This reflects the app manifest that declares the function.
- "applicationDefinition": This stores the object type, name, and ID.
- "startTime" and "endTime": These parameters are used to calculate the time elapsed within the function.

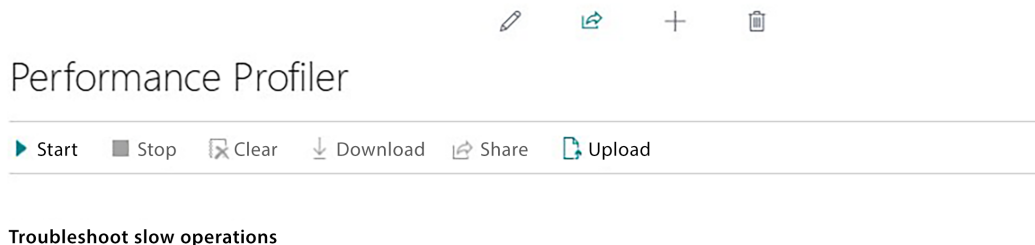
And this completes the example node!

This syntax mimics the standard F12 Chromium Performance Profiler that you might find in the Google Chrome or Edge Chromium browsers. To find out more about it, check out: <https://chromedevtools.github.io/devtools-protocol/tot/Profiler/>

Now that we know how to collect an AL profiler trace, what information it stores, and how to analyze it using Visual Studio Code, we could also add that the same profiler trace collection can be collected by users using the client. This has the advantage that it could also be analyzed directly in the UI.

To showcase how to collect an AL profiler trace similar to the one collected with Visual Studio Code, perform the following steps in the UI:

1. Run the browser client and search for Performance Profiler. Alternatively, you can click on the top-right breadcrumb question mark symbol (?) and select **Help & Support**, then click the **Analyze Performance** link in the **Troubleshooting** section. This action will open another browser's window with the in-client performance profiler page.



Use the Start and Stop buttons to record a process that you think is slow. The charts will show how much time each app used.

Figure 10.22: Performance Profiler

2. Click **Start**.
3. Go back to the previous browser window and search for the *Item Ledger Entry Analysis* report (the one we created and deployed in *Chapter 8, Report Development*), then click the **Print** button. When the report output is generated, you can close it and go back to the in-client profiler page in the other browser window and click **Stop**. This will trigger the finalization of the AL profiler trace that will be handled server-side via the standard AL code and the page will be updated with its output, like the following, where it shows the extensions that were active when collecting the trace and the time spent (in ms) running the code in each of them:

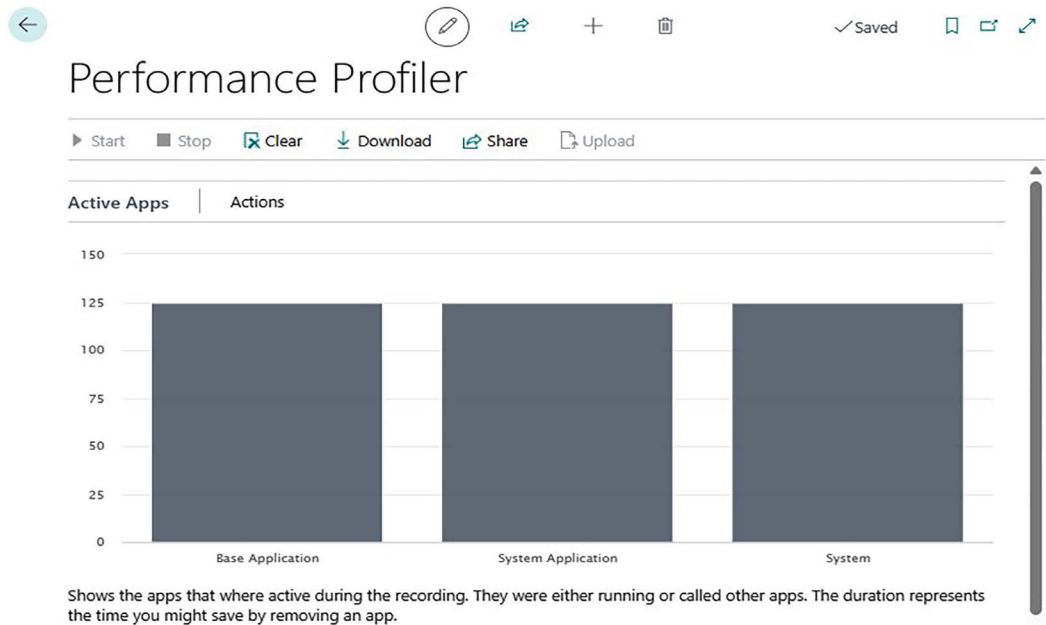


Figure 10.23: Data shown by Performance Profiler

4. By toggling the **Show Technical Information** button, we can enable three different performance analysis views:
  - **Time Spent:** This shows a graph of the self-time spent, aggregated by application name or publisher. This gives important information to determine, for example, whether the performance problem lies in the standard code (published by Microsoft) or a specific third-party extension.
  - **Time Spent by Application Object:** This provides a list of the most expensive object in terms of time consumption. It is possible to drill down into each of these objects to get a detailed analysis of every single function call and the time spent. This view could provide good insights into which objects and functions a potential performance problem might be nested in.

- **Call Tree:** This is an indented tree view that represents the top-down sequence of calls. This is probably the most important view of all and is similar to the profiler view offered by Visual Studio Code and helpful in identifying the application code flow and its performance bottlenecks.

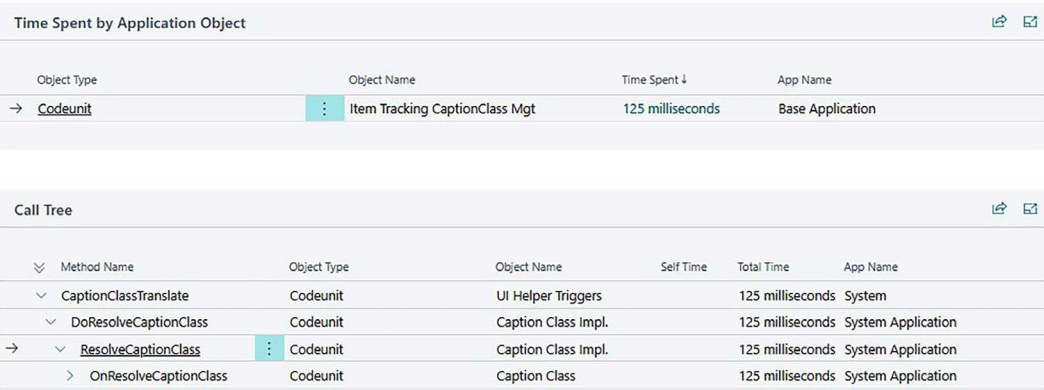


Figure 10.24: Time Spent and Call Tree performance analysis views

The **Performance Profiler** page also allows you to locally download the trace file just collected or share it through OneDrive. In this way, any user could collect a profiler trace on a relevant business process that is not performing as expected and, once the trace collection is complete, upload the profiler trace file for further inspection by developers, or even by Microsoft support if and when needed.

There is currently one very important difference, as you might already have noticed, between the AL profiler trace collection in Dynamics 365 Business Central online using the UI versus the trace collected through Visual Studio Code.

In the example provided, we perform the same data collection task (running the Item Ledger Entry Analysis report) on both the UI and Visual Studio Code, hence you might expect a higher number of object rows when running the in-client performance profiler trace collection, instead of just one (the “Item Tracking CaptionClass mgt” codeunit) with a self-time of 125 milliseconds and consequently more entries on the **Call Tree** sub-page, as happened when collecting the AL profiler trace using Visual Studio Code.

This difference in data collection is related to the fact that in Dynamics 365 Business Central online, the in-client performance profiler belongs to a sampling profiling type with an interval constant set to 100 milliseconds.

Note that since Dynamics 365 Business Central 2023 Wave 2 (version 23), it is possible to choose between 3 different intervals: 100 (this is the default), 150, or 50 milliseconds. It is typically a best practice to use the lowest sampling boundary (50 milliseconds) to collect as much data as possible to get the most accurate picture of the whole code execution.

To know more about the difference between the three and the impact on the trace output, read the following blog post: <https://www.waldo.be/2022/09/30/setting-the-sampling-interval-for-in-client-performance-profiling-business-central/>

And now, let's add the cherry on top of the cake.

To design the in-client performance profiler, the Microsoft Dynamics 365 Business Central application team requested that the technical team expose specific AL APIs (statements) to collect profiler traces via AL code. They then used these statements to build the **Performance Profiler** page and the main standard code that performs this task is contained in the single instance codeunit 1924 Sampling Performance Profiler. The following list details the exposed function signatures and their descriptions:

```
Start()
Starts AL profiler trace collection.
Stop()
Stops AL profiler trace collection.
IsRecordingInProgress(): Boolean
```

Then we can check if the performance profiling is already active server side.

```
GetData(): InStream
Gets the performance profiling data, in JSON format, after the recording has
been stopped.
SetData(ProfilingResultsInStream: InStream)
Sets the performance profiling data, in JSON format, from a stream.
GetProfilingNodes(var ProfilingNode: Record "Profiling Node")
Gets the results of the performance profiling per single object (node).
GetProfilingCallTree(var ProfilingNode: Record "Profiling Node")
```

This code gets the results of the performance profiling in a typical tree structure.

It is possible to directly reference this codeunit in a custom extension and create your own performance profiling collection tool, tailored to your specific needs. To provide you with an example, let's try to gather a very similar sampling trace to what we collected using the in-client profiler:

1. Create a new extension with a dependency on the app that contains the "Item Ledger Entry Analysis" report. See the following prototype of the app.json file:

```
{
  "id": "fbfd3d52-75a2-49f3-b081-31a165250f8c",
  "name": "AL Performance Profiler Tool",
  "publisher": "PACKT",
```

```

    "brief": "Performance Profiler Tool to collect traces through UI",
    "description": "Performance Profiler Tool to collect traces through
UI",
    "version": "1.0.0.0",
    "privacyStatement": "",
    "EULA": "",
    "help": "",
    "url": "http://www.duiliotacconi.com",
    "logo": "./Logo/ExtLogo.png",
    "dependencies": [
      {
        "id": "dd03d28e-4dfe-48d9-9520-c875595362b6",
        "name": "Packt Demo Extension",
        "publisher": "PACKT",
        "version": "1.0.0.0"
      }
    ],
    "screenshots": [],
    "platform": "1.0.0.0",
    "application": "23.0.0.0",
    "idRanges": [
      {
        "from": 50100,
        "to": 50149
      }
    ],
    "resourceExposurePolicy": {
      "allowDebugging": true,
      "allowDownloadingSource": true,
      "includeSourceInSymbolFile": true
    },
    "runtime": "12.0",
    "features": [
      "NoImplicitWith", "TranslationFile"
    ],
    "target": "Cloud"
  }
}

```

2. Go to the Command Palette (*Ctrl + Shift + P* or *F1*) and download the symbols.
3. Remove the `HelloWorld.al` file and create a folder called `src` with two subfolders: `codeunit` and `pageextension`.

4. In the codeunit subfolder, create a new file named PKTProfilerManagement.codeunit.al and add the following code:

```
codeunit 50111 "PKT Profiler Management"
{
    SingleInstance = true;

    procedure Start()
    var
        FileName: Text;
        SamplingInterval: Enum "Sampling Interval";
    begin
        if (SamplingPerformanceProfiler.IsRecordingInProgress()) or
            (Session.CurrentExecutionMode = ExecutionMode::Debug) then
            exit;

        //Collect sample every 50ms
        SamplingInterval := SamplingInterval::SampleEvery50ms;
        SamplingPerformanceProfiler.Start();

        //Do your stuff
        Report.run(Report::"Item Ledger Entry Analysis",false);

        SamplingPerformanceProfiler.Stop();

        FileName := 'TraceCollection.alcpuprofile';
        DownloadFromStream(SamplingPerformanceProfiler.GetData(),'', '',
            '', FileName);

    end;

    var
        SamplingPerformanceProfiler: Codeunit "Sampling Performance
        Profiler";
}
```

This codeunit object checks that the session is neither running in debug mode nor that another performance profiler trace is active under the hood. If this requirement is fulfilled, then it starts the profiler trace collection and the codeunit executes (in this simple example, it runs a report). Then it stops the trace collection, and through the GetData procedure it streams the JSON trace file to the browser client.

5. In the pageextension subfolder, create a new page extension named PKTItemLedgerEntriesExt.al and add the following code to execute the profiler trace collection through the UI:

```

pageextension 50113 "PKT ItemLedgerEntriesExt" extends "Item Ledger
Entries"
{
    actions
    {
        addlast(Creation)
        {
            action(PKTCollectALProfilerTrace)
            {
                Caption='Collect Profiler Trace';
                Image=ProfileCalendar;
                Promoted=true;
                PromotedIsBig=true;
                PromotedCategory=New;
                ApplicationArea = All;

                trigger OnAction()
                var
                    ProfilerManagement : Codeunit "PKT Profiler
Management";
                begin
                    ProfilerManagement.Start();
                end;
            }
        }
    }
}

```

6. Build the extension (*Ctrl + Shift + B*) and publish it without debugging (*Ctrl + F5*).
7. Search for the Item Ledger Entries list page.
8. Click on **New | Collect Profiler Trace** and let the report run behind the scenes.
9. When the TraceCollection.alcuprofile file is downloaded, move it into the .snapshots folder in the Visual Studio Code project and double-click it. You should have a sampling profiler trace something like the following:

TraceCollection.alcpuprofile X

Filter functions or files, or start a query()

As

.\*

Self Time in ms	Total Time in ms	Function Name
0.00	0%	1,427.21 100%
880.34	61%	1,427.21 39%
0.00	100%	0.00 0%
0.00	0%	218.71 100%
0.00	0%	218.71 100%
218.71	100%	218.71 0%
0.00	0%	328.15 100%
109.42	100%	109.42 0%
218.73	100%	218.73 0%

Figure 10.25: Profiler trace data for a file

Please note that your profiler trace file most probably will not have the same number and type of functions, because these could run faster than 50 milliseconds, hence they will not be collected by the sampling profile type.

Another reason might be that you have a different pool of extensions subscribing to extension publishers in the system and/or base application.

## Summary

In this chapter, we have learned how to run the classic debugger and we became familiar with the most important components of its interface. We also inspected some cool asynchronous debugging features that make our development lives easier, such as snapshots and performance profiling traces.

We have also seen how to pin non-debuggable functions and variables to avoid showing private data, to cope with privacy rules.

Now you’re ready to debug extensions, catch runtime errors while inspecting code flows, uncover performance bottlenecks, and deeply analyze AL code.

In the next chapter, we will master other asynchronous debugging techniques and more, by digging into the deep sea of telemetries.

## Leave a review!

*Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.*



*\*Limited Offer*

# 11

## Telemetry

In Dynamics 365 Business Central's October 2018 release, major version 15.x, Microsoft implemented a framework to send diagnostic traces (signals) asynchronously emitted by the platform and the application. Signals are collected in external repositories in the Azure Monitor collector called Application Insights. The collection of all signals is typically referred to as **telemetry**.

Initially, telemetry was implemented solely for the purposes of reactively troubleshooting cloud issues arising from the application or the platform. Release after release, Microsoft has added a lot of new signals or improved existing ones to collect all the information needed to:

- **Proactively** understand how an application is used and how it is performing (also known as **observability**).
- **Reactively** review platform and/or application execution data to determine the root cause of specific issues or performance bottlenecks.

In this chapter, you will learn about and master everything you need to know about telemetry to be able to use it to get insights from the application. We will deep dive into:

- Signal fundamentals
- How to enable telemetry in Dynamics 365 Business Central online
- Log analysis
- An Application Insights overview and its capabilities
- Power BI telemetry
- How to create custom signals

### Signal fundamentals

Telemetry signals can be defined as information that is collected at runtime during specific platform or application events and sent externally to an ingestion point to be analyzed individually or collectively. We'll go through an example later where this definition will become clearer.

You can get an overview of telemetry in the official documentation here:

<https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/telemetry-overview>

Let’s look at some essential background knowledge to get a fuller understanding of telemetry. In the following sections, we’ll cover dimensions, aggregated signals, and Microsoft’s recommendations, and run through a telemetry example.

## Dimensions

Information is sent in JSON format; every single JSON element is called a **dimension**. Dimension naming conventions follow the camel case notation. Dimensions can be grouped as:


- **General** (or core or generic): Common information that you will find in every signal, such as, for example, `message` or `severityLevel`. Some of this generic information can or cannot be found in every signal. The dimension name is the same, but it could be meaningless in some signals, forbidden, or simply not implemented, hence you might not find it. For example, `userId` can be found only in some signals but not all.
- **Custom**: These store peculiar and specific information for a given signal. These custom dimensions typically have a common set of environmental information details, for example, `aadTenantId`, `environmentType`, or `environmentName`, and a set of specific information, for example, `sqlStatement` in traces containing long-running queries or `alRecordsDeleted` in traces related to the applied retention policy.

Let’s now analyze how an authorization signal is structured with dimensions. The example we will look at now can be found through this link: *Authorization succeeded (Open Company)*, <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/telemetry-authorization-trace#authorization-succeeded-open-company>.

Every signal has a common dimension used as a unique identifier, called `eventId`. For example, for **Authorization succeeded (Open Company)**, the ID is `RT0004`. Checking the `eventId` for standard signals, you might notice that the first two characters (affixes) typically resemble the emitting component or a specific event within an entity lifecycle:

Affix	Name	Examples
AL	Application	Email sent, onboarding events, permission set assigned
CL	Client	Page open statistics, error feedback, client action invoked
LC	Lifecycle	Company creation or deletion, extension deployment
RT	Runtime	Performance, authorization, session management

Table 11.1: *eventId* affixes



You might find some **Application** signals in the **Lifecycle** section. This is caused by the prioritization of certain affixes over others when emitting the source affix as a signal `eventId`. A typical example of this is when job queue signals emitted by the application (which are AL signals) pair with the lifecycle ones related to scheduled tasks emitted by the server runtime (LC signals).

The following table contains some relevant RT0004 elements divided into general and custom dimensions with examples:

General Dimensions		Custom Dimensions	
timestamp	2023-05-02T14:17:16.0647956Z	aadTenantId	72fdeebd-28e2-4685-8494-c1a417aa6d59
message	Authorization Succeeded (Open Company)	environmentType	Production
severityLevel	1	environmentName	Production
session_Id	24e70257-c990-4e55-bcf4-f7d3a9319307	companyName	CRONUS UK Ltd.
user_id	a5c8b060-82d0-485a-9e90-4c230b56b95f	componentVersion	23.0.53597.56043
		eventId	RT0004
		clientType	Background
		serverExecutionTime	00:00:00.0124056
		sqlExecutes	3
		sqlRowsRead	14

Table 11.2: Runtime element dimensions

Signals could carry information that is worth checking alone. For example, `serverExecutionTime` represents the time spent by the service tier opening the company (as in, when a user tries to sign in to a company). In this case, it is quite fast, but there might be scenarios in which this time could increase by several seconds. It is worth checking how many `sqlExecute` signals are issued when opening the company, in case there is a latency problem, and how many `sqlRowsRead` signals are sent during this operation, in case there is a missing index or an issue with an application that scans a big table.

But the best expression of signals is given when aggregating information.

### Aggregated signals

For example, it is possible to summarize all signals for that specific `user_id` in a time frame or even just limited to a specific `session_id`.

Some signals have been designed to be grouped by application features and analyzed altogether, such as permission sets: different signals are emitted if a permission set was added or removed, it was assigned to or removed from a user or user group, or it was changed.

Some signals are emitted by the platform considering events that define an entity lifecycle. A couple of examples are extension deployment and opening a company. Extension-related signals are emitted when:

- Publishing an extension
- Compiling an extension
- Synchronizing an extension
- Installing an extension
- Updating an extension
- Uninstalling an extension
- Unpublishing an extension

In short, telemetry analyzes all deployment phases. This is useful to observe how often an extension is deployed and how well it performs. This is known as using telemetry as a source of **observability**. Alternatively, telemetry can be used as a source of **troubleshooting**. An example of that might be finding out when all data was erased from a sandbox by a deployment performed in clean mode.

The same principle could be applied to company lifecycles. These can be created, renamed, or deleted. All these operation events are instrumented with telemetry to keep track of them.

## Microsoft's recommendations

There are a few other points that Microsoft recommends are considered when it comes to telemetry:

- **Signals are (and should be) documented and actionable:** Telemetry signals might resemble API contracts, and Microsoft carefully maintains standard documentation related to signal definitions. You can find all the necessary information related to every single telemetry signal in the official documentation. These can be grouped by ID: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/telemetry-event-ids>.

They can also be grouped by area of interest: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/telemetry-area-overview>.

There is also a huge amount of example telemetry to get started with in this official GitHub repository: <https://github.com/microsoft/BCTech/tree/master/samples/AppInsights>.

- **Signals do not (and should never) contain any Personally Identifiable Information (PII):** Standard Microsoft signals do not contain any kind of PII, and where they might report some kind of correlation to a resource, the information is anonymized. An example is `user_Id`, which is a **Globally Unique Identifier (GUID)** used in several signals to help the customer correlate a Dynamics 365 Business Central user ID within specific activities. Only those who have access to the customer user list within the database can interpolate the GUID with a specific user record.

- **Signals are sent asynchronously:** The instrumentation of telemetry is done through separate channels of communication, hence they typically have a very low impact on the application or platform performance. Consider also that signals available to partners for ingestion are just a very small part of the telemetry emitted by Microsoft components and their asynchronous implementation is tested to be robust in the most stretched scenarios.

## Example

It should now be clear what signals are and their purposes, so let us take a couple of practical steps to get our hands dirty with telemetry. We will take as an example lifecycle telemetry signals related to job queues to keep them monitored. Use the following steps to reproduce the scenario:

1. Log in to your Dynamics 365 Business Central environment.
2. Search for **Job Queue Entries** and click **New**.
3. Create a recurring job queue entry that executes report 111 Customer Top 10 list every 2 minutes. See below the relevant setup parameters:
  - **Object Type to Run:** Report
  - **Object ID to Run:** 111
  - Enable the **Recurring Job** and all **Run on ...** checkboxes
  - **No. of Minutes between Run:** 2
4. Set **Status** to **Ready**.

In this way, the application will run a report every 2 minutes every day, and by enabling telemetry in this environment, we will be able to keep monitoring the job queues and scheduled tasks' lifecycle signals.

## Enabling partner telemetry in Dynamics 365 Business Central online

Microsoft is always collecting tons of telemetry data from the platform and application. Some of this telemetry data could also be enabled to be sent to a specific partner or customer's Application Insights ingestion points. Enabling telemetry is straightforward and is basically done in two easy steps that will take no longer than 10 minutes:

1. Create or define an Azure Application Insights resource.
2. Copy and paste the connection string to that resource on the tenant admin center environment card page.

Let us perform these tasks together:

1. Connect to your Azure subscription.
2. Search for **Application Insights** and click on **New**.

3. Provide the following details related to:

- Project (Azure subscription and resource group)
- Instance (name and region)
- Workspace (Azure subscription and Log Analytics workspace)

See below for an example for a new Application Insights resource:

**PROJECT DETAILS**

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ

Resource Group \* ⓘ  [Create new](#)

**INSTANCE DETAILS**

Name \* ⓘ

Region \* ⓘ

Resource Mode \* ⓘ ☐ Classic ☒ Workspace-based

**WORKSPACE DETAILS**

Subscription \* ⓘ

Log Analytics Workspace \* ⓘ

Figure 11.1: Sample Application Insights resource

4. Click **Review + Create** and then **Create**.
5. Resource deployment should take less than 1 minute, typically. When the deployment has completed, click on **Go to Resource overview**.
6. From the **Overview**, in the **Essentials** tab, copy the connection string that you'll find on the right-hand side. The connection string should be like the following:

```
InstrumentationKey=8a3f9bae-4c29-443e-9116-b67ad7d54544;IngestionEndpoint=https://westeurope-5.in.applicationinsights.azure.com/;LiveEndpoint=https://westeurope.livediagnostics.monitor.azure.com/
```

7. Open the Dynamics 365 Business Central tenant admin center and locate the environment where you would like to enable sending telemetry to the newly created Application Insights resource.



Dynamics 365 Business Central and Application Insights resources do not necessarily need to be in the same tenant, nor in the same data region. They could be separated. If you are managing your customers, you could typically decide to create an Application Insights resource on your tenant and send a bill with the ingestion cost to your customers separately.

8. Click on the **Define** hyperlink in the Application Insights **Connection String** field.
9. A new tab will open on the right. Toggle **Enable Application Insights** and paste the connection string copied previously.

Enable Application Insights



Connection String \*

InstrumentationKey=8a3f9bae-4c29-443e-9116-b67ad7d54544;Ingest...

Figure 11.2: Toggling Application Insights

10. Click **Save** to make the changes effective.



It is particularly important that you perform this action during non-working hours or when no-one is connected or there are no background running procedures. The connection string is saved in the tenant database, and to be loaded in memory and let the service start sending telemetry, a tenant **restart** is needed. All sessions will be closed during restart, hence if users are connected, they will be kicked off the application and requested to log in again.

When a restart is completed (it should typically take less than 1 minute), telemetry is sent to the ingestion point and the tenant admin center environment card should look like the following:

Application Insights Connection String (?)  
 InstrumentationKey=8a3f9bae-\*\*\*\*-\*\*\*\*-  
 \*\*\*\*\_  
 \*\*\*\*\*;IngestionEndpoint=https://w  
 esteurope-  
 5.in.applicationinsights.azure.com/LiveEn  
 dpoint=https://westeurope.livediagnosti  
 cs.monitor.azure.com/ ( [Modify](#) )

Figure 11.3: Tenant admin center environment card

And that is all. We are done with the setup. Let us double-check that everything is fine by running a simple query in Application Insights and see if the records are ingested:

1. In the **Application Insights** resource bar on the left, locate the **Monitoring** tab and click on **Logs**. If this is the first time you are accessing the **logs** pane, a welcome page is opened.

2. Close the welcome page if it pops up, and in the query window (sometimes known as the whiteboard), type traces and click the **Run** button, as shown below.

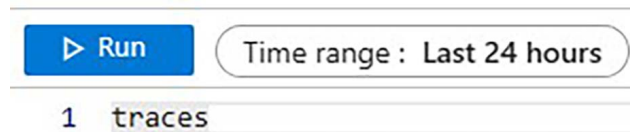


Figure 11.4: Running traces for a list of records

You should now see a list of records in the **Results** view, like the following:

Results

Chart

timestamp [UTC]	message	severityLevel	itemType	customDimensions
> 5/2/2023, 9:04:45.641 AM	Authorization Succeeded (Pre Open Company)	1	trace	{"guestUser":"False","entitlementSetIds":"3f2afeed-6fb5-4bf...
> 5/2/2023, 9:05:01.771 AM	Operation exceeded time threshold (SQL query)	2	trace	{"extensionVersion":"21.5.53619.56002","extensionId":"437d...
> 5/2/2023, 9:05:03.115 AM	Task 99d93d67-becd-4b4f-b4a5-04ef39ea1bd8 created: 455...	1	trace	{"notBefore":"2023-05-02T09:05:18.1150000Z","taskId":"99d...
> 5/2/2023, 9:05:04.052 AM	Authorization Succeeded (Open Company)	1	trace	{"result":"Success","parentSessionId":"NaN","component":"D...
> 5/2/2023, 9:05:09.287 AM	Authorization Succeeded (Open Company)	1	trace	{"result":"Success","parentSessionId":"NaN","component":"D...
> 5/2/2023, 9:05:09.302 AM	Authorization Succeeded (Open Company)	1	trace	{"result":"Success","parentSessionId":"NaN","component":"D...
> 5/2/2023, 9:05:13.349 AM	Authorization Succeeded (Open Company)	1	trace	{"result":"Success","parentSessionId":"NaN","component":"D...
> 5/2/2023, 9:05:13.349 AM	Authorization Succeeded (Open Company)	1	trace	{"result":"Success","parentSessionId":"NaN","component":"D...
> 5/2/2023, 9:05:13.365 AM	Web service called (ODataV4): MS/ODataV4/Company	1	trace	{"endpoint":"MS/ODataV4/Company","httpMethod":"GET","...

1s 345ms

Display time (UTC+00:00) ▾

Figure 11.5: List of records produced by traces

Congratulations, you are now ready to perform your first analysis of what is logged from your environment in terms of telemetry data! To do so, you will need to master a query language called **Kusto Query Language (KQL)**.

## KQL log analysis

KQL is a case-sensitive language, and it is very similar to **Transact-SQL (T-SQL)**. You can run single statements or organize them in a query by specifying a pipe sign (|) between each single operator.

You could consider your Application Insights resource as your database and signals as stored in tables in this database. If you look at the left side of the **Logs** whiteboard, you will notice a list of tables. There are two where Dynamics 365 Business Central sends signals: **pageViews** and **traces**. **pageViews** collects information related to pages opened within the environment, while the rest of the signals are all collected in the **traces** table.

Double-click on **pageViews**; it will add the table name on the whiteboard so that you can run it as a KQL query statement returning the entire table record set.

If you expand any of these two tables, you will see the metadata structure definition for the ingestion tables and their column data types.

Application Insights

- traces
  - timestamp (datetime)
  - message (string)
  - severityLevel (int)
  - itemType (string)
  - customDimensions (dynamic)

Figure 11.6: Column data types

There is plenty of documentation related to the KQL language. As a starting point, it is suggested to start with a simple tutorial from the official documentation: *Kusto Query Language (KQL) overview*, <https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query/>.

Once you have completed at least the three main tutorials (Learn common operators, Use aggregation functions, and join data from multiple tables), then we can have a look at a real-life scenario with job-queue-related telemetry. We'll start by isolating the relevant signals for a job queue entry. Such signals are sent when:

- Enqueued (AL0000E24)
- Started (AL0000E25)
- Finished with success (AL0000E26) or failure (AL0000HE7)

Signals are sent by the application with code unit 1351 Telemetry Subscribers and subscribe to the following publishers:

Event ID	Code Unit	Publisher Name
AL0000E24	453 "Job Queue Enqueue"	OnAfterEnqueueJobQueueEntry
AL0000E25	448 "Job Queue Dispatcher"	OnBeforeExecuteJob
AL0000E26	448 "Job Queue Dispatcher"	OnAfterSuccessHandleRequest
AL0000HE7	450 "Job Queue Error Handler"	OnBeforeLogError

Table 11.3: Signal publishers for a job queue entry

Let us count how many of these events have been sent by the application and ingested within a specific period using the following query:

```
let aadTenantId = "72fdeebd-28e2-4685-8494-c1a417aa6d59";
let environmentType = "Production";
let environmentName = "Production";
traces
| where customDimensions.aadTenantId == aadTenantId
```

```

    and customDimensions.environmentType == environmentType
    and customDimensions.environmentName == environmentName
| where timestamp > ago(7d)
| where customDimensions.eventId in
("AL0000E24", 'AL0000E25', "AL0000E26", "AL0000HE7")
| extend jobQueueEventId = toString(customDimensions.eventId)
| summarize count() by jobQueueEventId
| extend noOfEvents = toint(count_),
    eventName = case(
        jobQueueEventId == "AL0000E24", "Enqueued",
        jobQueueEventId == "AL0000E25", "Started",
        jobQueueEventId == "AL0000E26", "Finished",
        jobQueueEventId == "AL0000HE7", "Failed",
        "UNKNOWN")
| project-away count_
| project jobQueueEventId, noOfEvents, eventName
| sort by jobQueueEventId asc

```

The result of the query execution should be like the following:

JobQueueEventId	noOfEvents	eventName
> AL0000E24	1580	Enqueued
> AL0000E25	1577	Started
> AL0000E26	1576	Finished

Figure 11.7: Signal publishers for a query execution



The number of events should be different since it depends on how long the job queue entries have been running, the amount of data in your database, and other factors.

This query could have been written in a less verbose mode, of course, but it has been intentionally designed to showcase the most popular KQL elements – statements, operators, and functions – in action. Let us have a look at them.

## Statements

- **let:** This is used to set a variable equal to a constant value or a function, or – in more complex scenarios – a view.
- **extend:** This is the tabular statement that holds the final output. The tabular statement, together with all the **let** statements, separated by a semicolon (;), defines the KQL query.

## Operators

- **where:** Filters a table based on one or more predicates separated by and/or Boolean operands
- **extend:** Creates calculated columns and appends them to the output
- **summarize:** Generates a new table that is an aggregation of the input table
- **project:** Selects the columns to include in the output
- **project-away:** Excludes columns from the output
- **sort:** Determines the output display order (ascending or descending)

## Functions

- **ago(<datetime>):** Filters a table based on one or more predicates separated by and/or Boolean operands.
- **tostring(<value>), toint(<value>):** Assigns the input to a string or an integer output. This is typically used when the input value has a dynamic data type, such as elements contained in the `customDimensions` array.

The ones used are just the most common when designing KQL queries. If you want an overview of what this powerful language offers, this is duly described, with examples, in the official documentation.

Let us go back to our example. Now that we know that our job queue is running just fine, let's double-check and correlate values with their technical equivalent: the scheduled tasks.

Every time a job queue is active, at some point, the application will trigger a `TaskScheduler.CreateTask()` AL statement that generates a new task (a platform artifact) to be run at a specific date and time in the service tier and through a background session, as described in **Task Scheduler**: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-task-scheduler>.

This decoupling between the application (job queue) and platform (Task Scheduler) sides is very important to understand since it is crystal clear that telemetry signals for the scheduled task will not be triggered by the application but by specific platform telemetry instrumentations. Signals are emitted when a task is in any of the following states:

- **Created (LC0040):** Triggered by `TaskScheduler.CreateTask`.
- **Set in the Ready state (LC0041):** Triggered by `TaskScheduler.SetReadyState`.
- **Removed (LC0042):** Triggered by `TaskScheduler.CancelTask`.
- **Completed (LC0043):** It ended successfully.
- **Canceled (LC0044):** Anything that would trigger the service to cancel the task session execution. For example, a background session killed from the Dynamics 365 Business Central tenant admin center or through a statement like `session.StopSession()`.
- **Failed (LC0045):** Anything that would raise an exception.
- **Timeout (LC0057):** Failed due to timeout.

Worth mentioning is, if you look at the common and custom dimensions for the scheduled task signals, you might notice that none of them have a reference to the source that generated the scheduled task. This is because they could be generated by any code that is implementing the TaskScheduler data type and not only by job queues.

With this information, we could first check how many events have been generated by running the following KQL query:

```
let aadTenantId = "72fdeebd-28e2-4685-8494-c1a417aa6d59";
let environmentType = "Production";
let environmentName = "Production";
traces
| where customDimensions.aadTenantId == aadTenantId
    and customDimensions.environmentType == environmentType
    and customDimensions.environmentName == environmentName
| where timestamp > ago(7d)
| where customDimensions.eventId in
("LC0040", 'LC0041', "LC0042", "LC0043", "LC0044", 'LC0045', "LC0057")
| extend taskEventId = tostring(customDimensions.eventId)
| summarize count() by taskEventId
| extend noOfTaskEvents = toint(count_),
    eventName = case(
        taskEventId == "LC0040", "Create",
        taskEventId == "LC0041", "Ready",
        taskEventId == "LC0042", "Removed",
        taskEventId == "LC0043", "Completed",
        taskEventId == "LC0044", "Canceled",
        taskEventId == "LC0045", "Failed",
        taskEventId == "LC0057", "Timeout",
        "UNKNOWN")
| project taskEventId, noOfTaskEvents, eventName
| sort by taskEventId asc
```

The result of the query execution should be like the following:

Results		Chart
taskEventId	noOfTaskEvents	eventName
> LC0040	1584	Create
> LC0042	1577	Removed
> LC0043	1581	Completed

Figure 11.8: Signal publishers for another query execution



The number of events should be different from what you see in the figure.

Then we could completely analyze the flow of one job queue entry and, in the end, reconcile all job queue entries with their equivalent scheduled task to understand if everything is running fine.

To analyze the flow of a job queue entry, we should take at least one job queue creation event (AL0000E24). Let's run this query to pick one up:

```
let aadTenantId = "72fdeebd-28e2-4685-8494-c1a417aa6d59";
let environmentType = "Production";
let environmentName = "Production";
traces
| where customDimensions.aadTenantId == aadTenantId
    and customDimensions.environmentType == environmentType
    and customDimensions.environmentName == environmentName
| where timestamp > ago(7d)
| where customDimensions.eventId == 'AL0000E24'
    and customDimensions.alJobQueueObjectType == "Report"
    and customDimensions.alJobQueueObjectId == "111"
| sort by timestamp asc
| project customDimensions.alJobQueueId, customDimensions.alJobQueueScheduledTaskId
| take 1
```

And the output should look like this:

Results		Chart
customDimensions_alJobQueueId		customDimensions_alJobQueueScheduledTaskId
> 6bdc3fb9-a4ab-4b9e-8a2a-446322edf80e		4ed2285e-a06a-405c-9bac-d9de2e911b61

Figure 11.9: Job queue creation event



The two GUIDs should be different from what you see in the figure.

Great. We now have a specific job queue entry (6bdc3fb9-a4ab-4b9e-8a2a-446322edf80e) that should have been created that carries the ID for a specific scheduled task creation (4ed2285e-a06a-405c-9bac-d9de2e911b61). We can then bind all the job queue entry signals together with the scheduled task ones and sort them by timestamp to understand the flow of signals:

```
let aadTenantId = "72fdeebd-28e2-4685-8494-c1a417aa6d59";
let environmentType = "Production";
let environmentName = "Production";
let jobId = "6bdc3fb9-a4ab-4b9e-8a2a-446322edf80e";
let scheduledTaskId = "4ed2285e-a06a-405c-9bac-d9de2e911b61";

traces
| where customDimensions.aadTenantId == aadTenantId
    and customDimensions.environmentType == environmentType
    and customDimensions.environmentName == environmentName
| where timestamp > ago(7d)
| where customDimensions.alJobQueueScheduledTaskId == scheduledTaskId
    or customDimensions.taskId == scheduledTaskId
| project timestamp, toString(customDimensions.eventId), message
| sort by timestamp asc
```

And the result should look like the following:

timestamp [UTC]	customDimensions_eventId	message
> 5/2/2023, 1:29:58.234 PM	LC0040	Task 4ed2285e-a06a-405c-9bac-d9de2e911b61 created: 448 scheduled to run after 2023-05-02 13:29:59Z. Ready to run: True
> 5/2/2023, 1:29:58.375 PM	AL0000E24	Job queue entry enqueued: 6BDC3FB9-A4AB-4B9E-8A2A-446322EDF80E
> 5/2/2023, 1:30:00.445 PM	AL0000E25	Job queue entry started: 6BDC3FB9-A4AB-4B9E-8A2A-446322EDF80E
> 5/2/2023, 1:30:03.820 PM	LC0042	Task 4ed2285e-a06a-405c-9bac-d9de2e911b61 removed: 448
> 5/2/2023, 1:30:03.835 PM	LC0043	Task 4ed2285e-a06a-405c-9bac-d9de2e911b61 main codeunit 448 completed.

Figure 11.10: List of job queue entry signals

Diving deep into the timeline, you can clearly see that enqueueing a job queue entry (AL0000E24) correspondingly creates a scheduled task (LC0040) immediately. When hitting the scheduled time, the job starts (AL0000E25), and it runs until it is finished. When the job is finished, the scheduled task is removed (LC0042) and returns the user to the application with a successful completion message (LC0043). If you check the difference in time between when the job started (1:30:00.445) and the task report was scheduled to be completed (1:30:03.835), we might say that this specific job took 3.39 seconds.

Let's see if there are other events within this job queue time frame – from enqueueing, 13:29:58 UTC, to completion, 13:30:03 UTC:

```
let aadTenantId = "72fdeebd-28e2-4685-8494-c1a417aa6d59";
let environmentType = "Production";
let environmentName = "Production";

traces
| where customDimensions.aadTenantId == aadTenantId
    and customDimensions.environmentType == environmentType
```

```

    and customDimensions.environmentName == environmentName
  | where timestamp between
    (todatetime('2023-05-02T13:29:58.234511Z') ..
    todatetime('2023-05-02T13:30:03.8359811Z') )
  | extend eventId = tostring(customDimensions.eventId)
  | project timestamp, eventId, message
  | sort by timestamp asc

```

The results should look like the following:

timestamp [UTC]	eventId	message	session_id
> 5/2/2023, 1:29:58.234 PM	LC0040	Task 4ed2285e-a06a-405c-9bac-d9de2e911b61 created: 448 scheduled t...	fe2e5c28-5193-4580-8540-7110265d690d
> 5/2/2023, 1:29:58.375 PM	AL0000E24	Job queue entry enqueued: 6BDC3FB9-A4AB-4B9E-8A2A-446322EDF80E	fe2e5c28-5193-4580-8540-7110265d690d
> 5/2/2023, 1:30:00.242 PM	RT0003	Authorization Succeeded (Pre Open Company)	N/A
> 5/2/2023, 1:30:00.257 PM	RT0004	Authorization Succeeded (Open Company)	ee0704d2-80d2-43b2-a160-a9885de4e536
> 5/2/2023, 1:30:00.445 PM	AL0000E25	Job queue entry started: 6BDC3FB9-A4AB-4B9E-8A2A-446322EDF80E	ee0704d2-80d2-43b2-a160-a9885de4e536
> 5/2/2023, 1:30:03.710 PM	RT0006	Report rendered: 111 - Customer - Top 10 List	ee0704d2-80d2-43b2-a160-a9885de4e536
> 5/2/2023, 1:30:03.820 PM	LC0042	Task 4ed2285e-a06a-405c-9bac-d9de2e911b61 removed: 448	ee0704d2-80d2-43b2-a160-a9885de4e536
> 5/2/2023, 1:30:03.835 PM	LC0040	Task e4a3a3a5-454a-45fd-b19a-698fd3fe2230 created: 448 scheduled to ...	ee0704d2-80d2-43b2-a160-a9885de4e536
> 5/2/2023, 1:30:03.835 PM	AL0000E24	Job queue entry enqueued: 6BDC3FB9-A4AB-4B9E-8A2A-446322EDF80E	ee0704d2-80d2-43b2-a160-a9885de4e536
> 5/2/2023, 1:30:03.835 PM	AL0000E26	Job queue entry finished: 6BDC3FB9-A4AB-4B9E-8A2A-446322EDF80E	ee0704d2-80d2-43b2-a160-a9885de4e536
> 5/2/2023, 1:30:03.835 PM	LC0043	Task 4ed2285e-a06a-405c-9bac-d9de2e911b61 main codeunit 448 comp...	ee0704d2-80d2-43b2-a160-a9885de4e536

Figure 11.11: Events in job queue time frame

RT0003 and RT0004 are typical runtime events related to session creation. In this case, if you analyze each event, you might notice that they are related to the creation of a new session (in this example: ee0704d2-80d2-43b2-a160-a9885de4e536). This new session runs in the background (clientType = Background) and represents the context in which the report will run.

RT0006 is the signal that will gather statistics about successful report rendering. To learn more about it, visit *Analyzing Report Telemetry*: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/telemetry-reports-trace>.

In this specific case, to generate the report, it took 2.97 seconds (totalTime). This means that all other operations related to creating the background session and removing the scheduled task when finished took  $3.39 - 2.97 = .42$  seconds. Not bad.

What if the job queue fails? Let's make it fail on purpose and see what's happening:

1. Log in to your Dynamics 365 Business Central environment.
2. Search for User Setup and, if it doesn't already exist, add the user that is running the job queue.
3. Set **Time Sheet Admin** to **Yes**.
4. In Visual Studio Code, create a new extension.
5. Name the extension, for example, Packt Make Report Fail.
6. Set up launch.json to connect to your environment and download symbols.

7. Create a code unit that subscribes to `OnAfterSubstituteReport` and raises an error if the user that is running the job queue is not a Time Sheet admin:

```
codeunit 50139 "PKT ReportSubscribers"

{
    [EventSubscriber(ObjectType::Codeunit, Codeunit::ReportManagement,
    'OnAfterSubstituteReport', '', false, false)]
    local procedure OnSubstituteReport(ReportId: Integer; var
    NewReportId: Integer)
    begin
        if ReportId = Report::"Customer - Top 10 List" then
            MakeReportFail();
    end;

    local procedure MakeReportFail()
    var
        UserSetup : Record "User Setup";
        TimeSheetAdminErr: Label 'You must be Time Sheet Admin to run
    this report.';
    begin
        If UserSetup.get(UserId) then
            if not UserSetup."Time Sheet Admin." then
                Error(TimeSheetAdminErr);
    end;
}
```

8. Generate the needed permission sets.
9. Build the extension and deploy it.
10. Repeat steps 1 to 3 but set **Time Sheet Admin** to **No**.

Since the recurring task is scheduled to run every 2 minutes, just wait for at least that time to be sure that a scheduled task will try to generate the report and fail. Then go back to the log analytics in your Application Insights resource and check if there are some failures in the job queues lifecycle using the first query that we have used in the paragraph.



Due to its asynchronous nature, signals from Dynamics 365 Business Central online could be sent within a delay of a few seconds.

The result should look like the following:

jobQueueEventId	noOfEvents	eventName
> AL0000E24	27	Enqueued
> AL0000E25	28	Started
> AL0000E26	27	Finished
> AL0000HE7	1	Failed

Figure 11.12: Checking for failures in the job queues lifecycle

The job queue is failing (AL0000HE7), and if you isolate that specific event, it is possible to get the scheduled task ID from its CustomDimension to be used in the second query to determine what is running behind the scenes. The AL0000HE7 signal contains the failing call stack trace (alJobQueueStacktrace) in its CustomDimension:

```
ReportSubscribers(CodeUnit 50139).MakeReportFail line 7 - Packt Make A Report Fail
by PACKT
\ReportSubscribers(CodeUnit 50139).OnSubstituteReport line 3 - Packt Make A Report
Fail by PACKT
\ReportManagement(CodeUnit 44).OnAfterSubstituteReport(Event) line 2 - Base
Application by Microsoft
\ReportManagement(CodeUnit 44).SubstituteReport line 2 - Base Application by
Microsoft
\"Reporting Triggers"(CodeUnit 2000000005).SubstituteReport(Event) line 2
\"Job Queue Start Report"(CodeUnit 487).ProcessSaveAs line 12 - Base Application
by Microsoft
\"Job Queue Start Report"(CodeUnit 487).RunReport line 37 - Base Application by
Microsoft
\"Job Queue Start Report"(CodeUnit 487).OnRun(Trigger) line 2 - Base Application
by Microsoft
\"Job Queue Start Codeunit"(CodeUnit 449).RunReport line 9 - Base Application by
Microsoft
\"Job Queue Start Codeunit"(CodeUnit 449).OnRun(Trigger) line 19 - Base
Application by Microsoft
\"Job Queue Dispatcher"(CodeUnit 448).HandleRequest line 28 - Base Application by
Microsoft
\"Job Queue Dispatcher"(CodeUnit 448).OnRun(Trigger) line 19 - Base Application by
Microsoft\
```

This concludes a simple exercise related to log analysis. We could write an entire book on how to apply KQL queries in real-life scenarios; now that you have learned and mastered the basics, the sky is the limit. You can take a look at other examples from the official GitHub repo: <https://github.com/microsoft/BCTech/tree/master/samples/AppInsights/KQL/Queries>.

Now that we have shown how to use KQL queries in a practical case and got our hands dirty with Application Insights, let's have a look at what else this suite offers in terms of features and create an alert against our failing job queue.

## Application Insights

As we mentioned at the start of the chapter, telemetry signals are collected in external repositories in an Azure Monitor collector called **Application Insights**. Application Insights offers a variety of monitoring tools to provide insights into the usage and performance of web applications. For Dynamics 365 Business Central, the top three most useful ones are the following:

- Alerts
- Application dashboards
- Workbooks

Once again, please note that these are just some of the many features that Application Insights offers, and since it is a large area of investment by Microsoft, we can expect even more enhancements in the future.

## Alerts

Some KQL query results might be useful to reactively send a communication that can be tailored to be informative for a particular persona. Such messages are typically an alert for something that has happened or is about to happen. Within Application Insights, it is possible to define alerts based on KQL query output. Let's consider the previous example and set up an alert that sends an email every time a job queue entry fails:

1. In the left action pane, in the **Monitoring** section, click on **Alerts**.



Figure 11.13: Alerts option on the left action pane

1. Click on **Create** and then **Alert Rule**.
2. Choose **Custom log search** for **Signal name**.

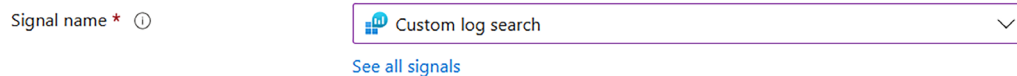


Figure 11.14: Naming a signal

3. Add the following **Search query**, which will take all failed job queue events in the past 10 minutes.

Search query \*

```
traces
| where customDimensions.aadTenantId == "72fdeebd-28e2-4685-8494-c1a417aa6d59"
  and customDimensions.environmentType == "Production"
  and customDimensions.environmentName == "Production"
| where timestamp > ago(10m)
| where customDimensions.eventId == "AL0000HE7"
```

Figure 11.15: Adding a query to a signal

- 4. Summarize the results by table rows with **Count** as **Aggregation type** and **Aggregation granularity** of **10 minutes**.
- 5. Set the logic to raise an alert when the result of the query is not empty, as shown below:

Operator \* ⓘ

Greater than

Threshold value \* ⓘ

0

Frequency of evaluation \* ⓘ

10 minutes

Figure 11.16: Setting query logic

Please note that changing **Frequency of evaluation** will influence how much this alert will cost per month. For example, a rule like the one above running every **10 minutes** will cost \$1 per month. Changing the frequency to, for example, **1 minute** will cost \$5 per month. The lower the frequency, the higher the cost will be.

Keep in mind that if you need to change the **Frequency of evaluation**, this must be aligned in the KQL query as well using the function `> ago()` to avoid missing or duplicate signals.

- 6. Click on the **Next: Actions >** button.
- 7. After defining the triggering condition, it is time to define the alert action or set of actions to be executed. This action or set of actions must be assigned to an action group first. Click on **New Action Group**.
- 8. Provide an **Action group name** and **Display name** such as, for example, **Job Queue**, and click on **Next: Notifications >**.
- 9. Decide what **Notification type** to implement and choose **Email/SMS message/Push/Voice** from the drop-down menu.

10. Give a **Name** to the notification type, for example, SendEmail, and in the right pane, toggle **Email** and provide a valid email address:

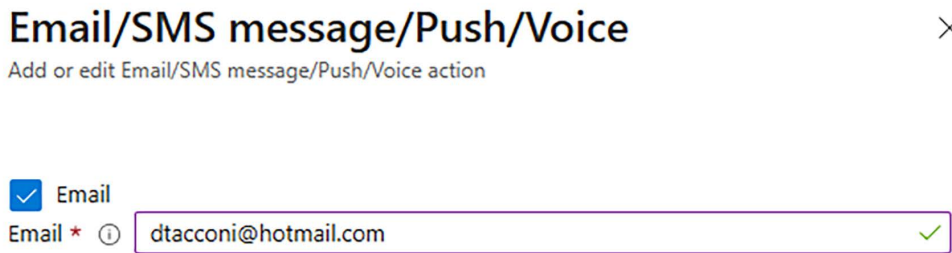
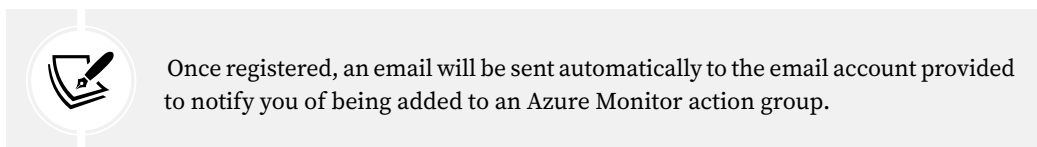


Figure 11.17: Providing an email address for SendEmail



11. At this point, we have already created an action group to send an email. There could be other opportunities to trigger further actions within this action group, such as running an Azure function or triggering a logic app to perform some actions proactively. But we are going deep into this, and we will let you self-explore all other action group capabilities. Just click **Review + Create** and **create** the action group.
12. Click on **Next: Details >**.
13. Provide the **Severity** of the event and change it to **1 – Error**; also, provide a name, such as **Job Queue Failed**, and a description like **Job queue has failed**.

We have then set an alert rule to send an email when a job queue is failing, and the alert should be in an **Enabled** state.

Name ↑↓	Condition	Severity ↑↓	Status ↑↓
<input type="checkbox"/> Job Queue Failed	Table rows > 0	1 - Error	✓ Enabled

Figure 11.18: Enabling an alert rule

Now it is time to test if this works. Log in to your Dynamics 365 Business Central environment and restart the job queue entry. This should fail almost immediately. Just wait up to 10 minutes and you should receive an email like the following:



**Fired:Sev1 Azure Monitor Alert Job Queue  
Failed on packtappinsightsresource ( mi-  
crosoft.insights/components ) at 5/12/2023  
3:53:22 PM**

[View the alert in Azure Monitor >](#)

**Summary**

Alert name	Job Queue Failed
Severity	Sev1
Monitor condition	Fired

Figure 11.19: Failure email notification

The email body will also contain the description, number of violations, and other useful or actionable information.

**Dashboards**

Since the advent of the Power BI telemetry report in the marketplace, directly provided by the Dynamics 365 Business Central server runtime team, creating application dashboards within Application Insights is no longer so common. Power BI visualization and what has been offered out of the box in terms of KPI analysis within this tool are more flexible and complete. The only point that remains in favor of Azure dashboards is that these are fully integrated into the Application Insights resource. In any case, for professional and proficient use, Microsoft recommends the uptake and use of Power BI telemetry reports. Within this book, we will just have a quick look at how to create simple dashboards.

After logging in to Application Insights, the **Overview** section is displayed first. Within this section, you have an essential snapshot of the resource properties (location, subscription, connection string, and others) and a few performance KPI graphs displayed at the bottom. In the upper-left corner, there is a button with a star icon called **Application Dashboard** that will redirect you to create or edit a fully customizable single-pane view of the most relevant query results within the specific Application Insights resource.

See a sample dashboard below:

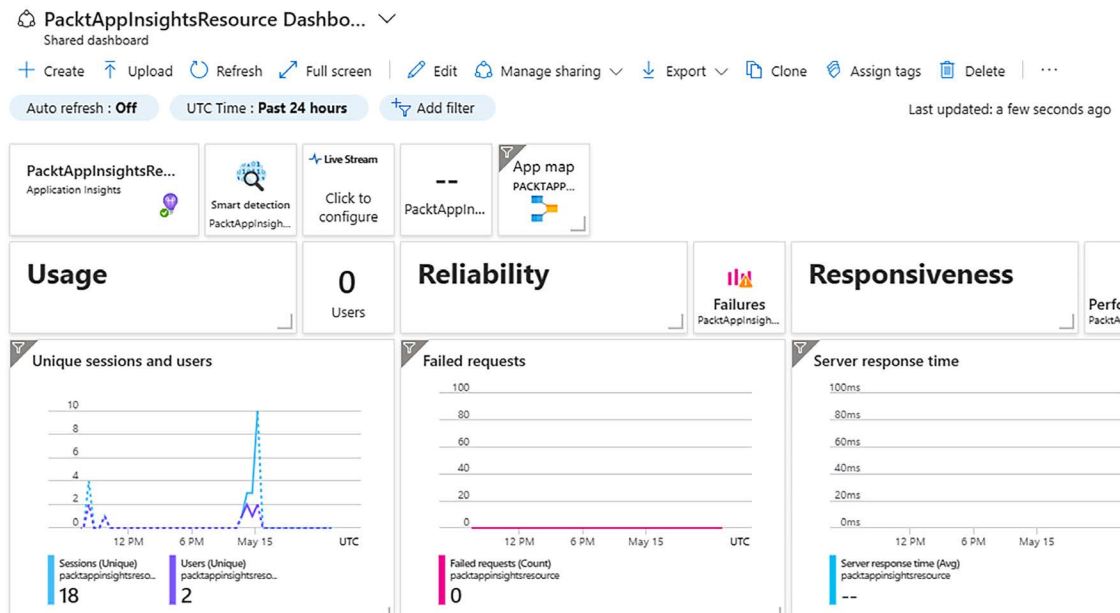


Figure 11.20: Dashboard of query results

It is possible to create new dashboards or change the existing ones through the UI, but the best – and fastest – way to show how to prepare and deploy dashboards is to analyze and implement the example provided in the standard Microsoft GitHub repository: <https://github.com/microsoft/BCTech/tree/master/samples/AppInsights/Dashboard>.

This repository contains an ARM template used to perform the dashboard deployment, and if you need to deploy your own dashboard, it is a good example to clone:

1. In the GitHub repository, click on the following button:



Figure 11.21: Deploy to Azure button on GitHub

The code behind this button is the following: <https://portal.azure.com/#create/Microsoft.Template/uri/https%3a%2f%2fraw.githubusercontent.com%2fmicrosoft%2fBCTech%2fmaster%2fsamples%2fAppInsights%2fDashboard%2fazuredploy.json>.

It will trigger an ARM template form.

2. In the ARM template form, add the project and instance details, such as the **Subscription ID**, **Resource Group Name**, and **Application Insights Name**:

**Instance details**

Region * ⓘ	(Europe) West Europe ✓
App Insights Subscription Id ⓘ	5abbf7c1- ✓
App Insights Resource Group Name ⓘ	appinsightresgroup ✓
App Insights Name * ⓘ	PacktAppInsightsResource ✓

Figure 11.22: Adding project details to the template form

3. Click **Review + create** and then **Create** to confirm.
4. Once deployment has been completed, open the **Application Insights resource**.
5. In the **Overview** section, click on **Application Dashboard**.
6. Drill down to inspect the available dashboards and choose, for example, **Sessions**, to see what the dashboard looks like:

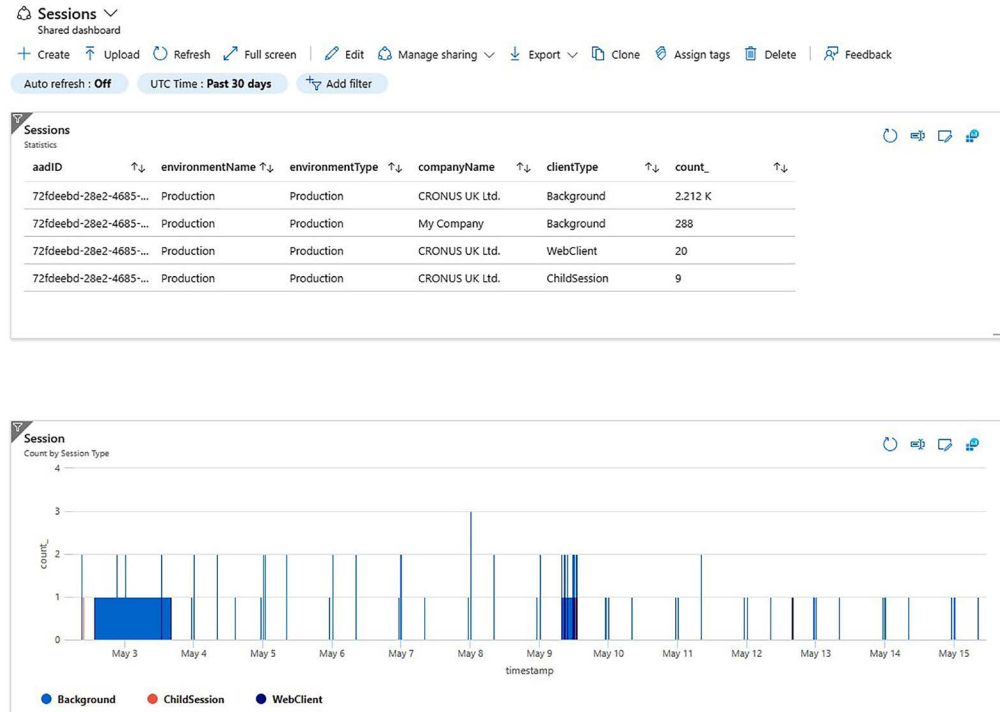


Figure 11.23: Custom dashboard

Congratulations! You have deployed a set of new dashboards. But how have these dashboards been created? In the **Sessions** dashboard, click **Export** and then **Download**. This operation results in the download of a file called `Sessions.json (<DashboardName>.json)`. If you inspect the file, you might notice that it contains several nodes and some of these are storing the KQL query used to gather data to be displayed. Try to change, for example, `"PartTitle": "Sessions"` into `"PartTitle": "My Sessions"` and click on the **Upload** button to import the changed file back.

Uploading the changed JSON file will create a new dashboard with a different title for the first chart.

If you do not want to clone and work with the sample provided in the GitHub repository, you could always create a new dashboard manually. Let's create a dashboard to summarize and display all job-queue-related events:

1. On the **application dashboard**, click on **Create** and select the **Custom** tile.
2. Give a name to the dashboard like **Job Queue** and save it.

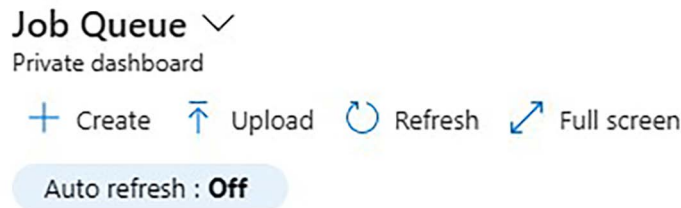


Figure 11.24: Job queue dashboard

3. Go back to the logs and execute the same query to get all job-queue-related events, but this time, without specifying a time range in the query:

```
let aadTenantId = "72fdeebd-28e2-4685-8494-c1a417aa6d59";
let environmentType = "Production";
let environmentName = "Production";
traces
| where customDimensions.aadTenantId == aadTenantId
    and customDimensions.environmentType == environmentType
    and customDimensions.environmentName == environmentName
| where customDimensions.eventId in
("AL0000E24", 'AL0000E25', "AL0000E26", "AL0000HE7")
| extend jobQueueEventId = tostring(customDimensions.eventId)
```

```
| summarize count() by jobQueueEventId
| extend noOfEvents = toint(count_),
    eventName = case(
        jobQueueEventId == "AL0000E24", "Enqueued",
        jobQueueEventId == "AL0000E25", "Started",
        jobQueueEventId == "AL0000E26", "Finished",
        jobQueueEventId == "AL0000HE7", "Failed",
        "UNKNOWN")
| project-away count_
| project jobQueueEventId, noOfEvents, eventName
| sort by jobQueueEventId asc
```

4. Once the query is executed, click on **Chart** in the result pane to display the result set as a visual chart.
5. Click on the right pane, **Chart Formatting**, and change **Chart type** to a stacked bar chart. The results should look like the following:

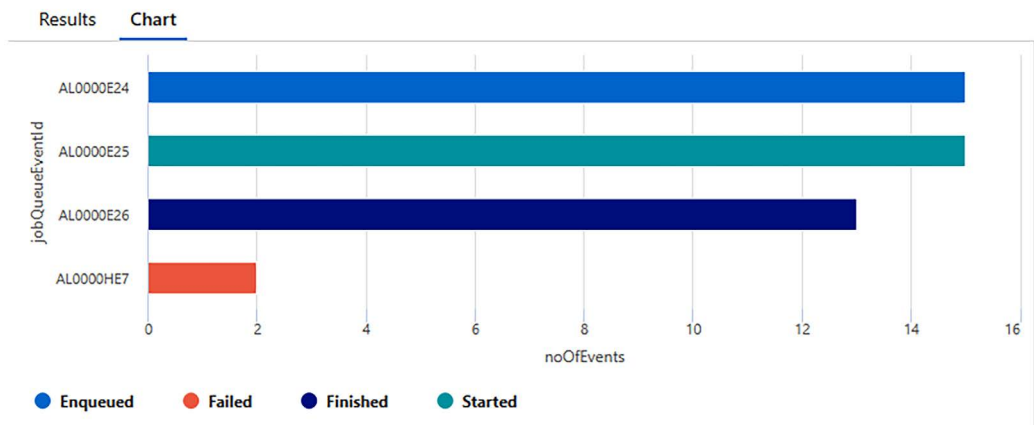


Figure 11.25: Setting a chart type

6. In the top menu bar, click **Pin to** and select **Azure Dashboard**.
7. Select the **Private** dashboard and **Job Queue** and click on **Pin**.

- Go back to the dashboards and inspect the dashboard just created.

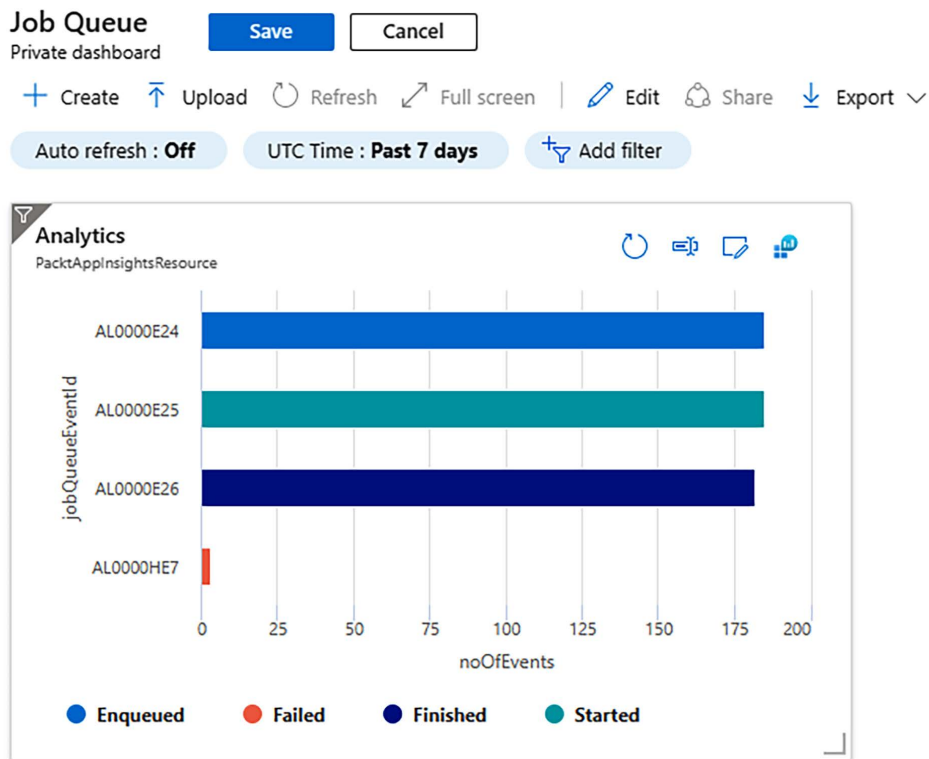


Figure 11.26: Adding a chart to a dashboard

The pinning action can not only be performed on an existing or brand-new Azure dashboard but also on a more modern Grafana dashboard. The implementation of this technology can be found in the blog post *Introducing Azure Managed Grafana for improving your Dynamics 365 Business Central telemetry views*: <https://demiliani.com/2022/09/02/introducing-azure-managed-grafana-for-improving-your-dynamics-365-business-central-telemetry-views/>.

## Workbooks

The Dynamics 365 Business Central product team is resolving platform, application, and performance issues exclusively through telemetry. Within every topic or feature analysis, they have identified a flow of documented verifications and data checks that might quickly point out a specific root cause. This set of information and investigation steps is typically defined as a **troubleshooting guide (TSG)** or workbook.

The same approach has been evangelized for partner telemetry, and Application Insights offers a similar workbook feature in its suite. Let's create a simple workbook considering the past example:

- Go to **Application Insights** and then to **Logs**.
- Execute the same query to get all job-queue-related events used to create the Azure dashboard.

3. In the top menu bar, click **Pin to** and select **Send to workbook**.
4. In the **New workbook** tab, choose **Azure Monitor** and check **Add a time range parameter and have generated steps reference it**, then click on **Send to workbook**.

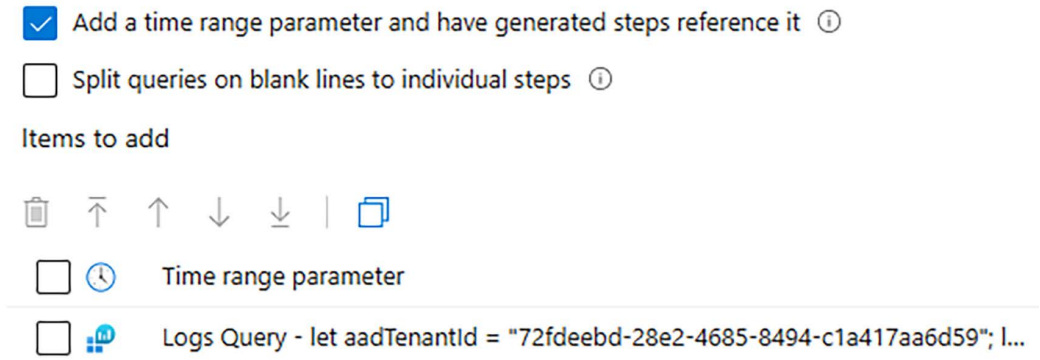


Figure 11.27: Configuring a new workbook

This action will generate a prototype workbook with a **Time range** control element. You can play around with the **Time range** dropdown to refresh the query result. This is what it should look like.

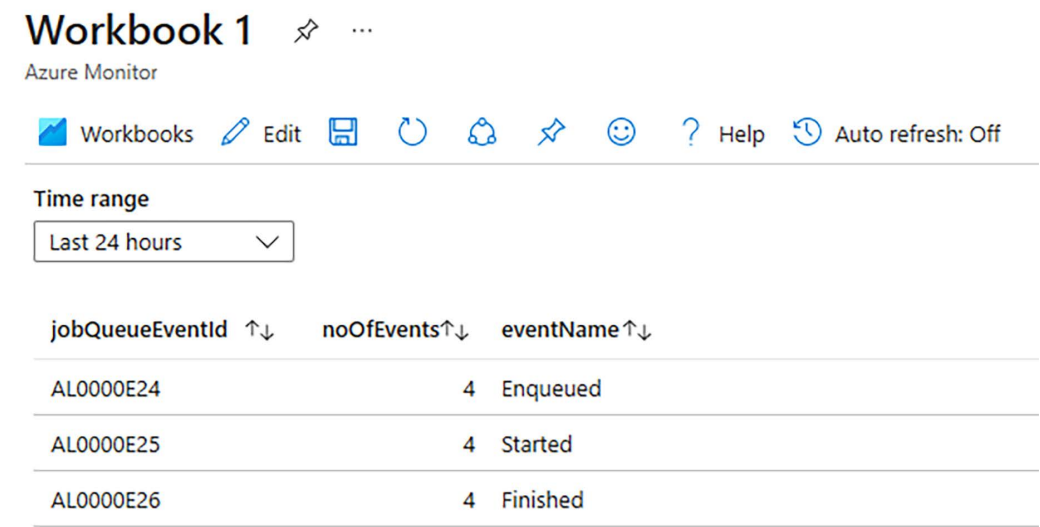


Figure 11.28: Layout of a new workbook

5. Click **Edit**.

6. On the right, you will see two **Edit** buttons, and on the left, an **Add** dropdown. The **Edit** buttons allow you to change the time range, query, or any dynamic control added and insert other elements above and/or below to, for example, define the query and its usage (what it is for, the expected results, and so forth). Click on the first **Edit** button at the top and then click **Add**:

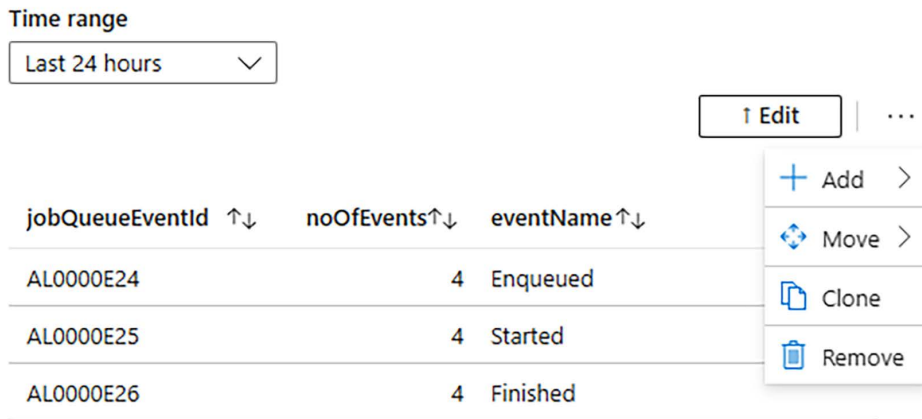


Figure 11.29: Editing columns in a workbook

7. You could now add to the workbook different types of controls (text, parameters, and queries, for example). Choose **Add text**:



Figure 11.30: Adding controls to a workbook

8. Add the following Markdown text:

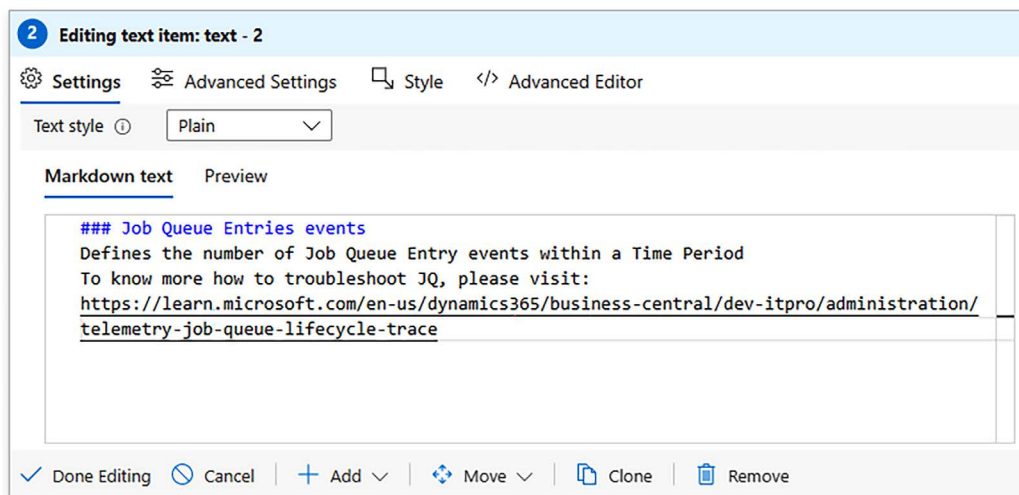


Figure 11.31: Adding Markdown text to a workbook

9. Click **Done Editing** for the text box control.
10. Click **Done Editing** again in the workbook menu bar, and then **Save as**.
11. Change the **Title** of the workbook from **Workbook 1** to **Job Queue Entries** and provide an appropriate location for the resource and click **Apply**.
12. Click **Done Editing** again in the workbook menu bar. The now-stored workbook should look like the following:

**Job Queue Entries** ✨ ...

Azure Monitor

Workbooks Edit Save Refresh Add Pin Share ? Help Auto refresh: Off

**Time range**

Last 24 hours

**Job Queue Entries events**

Defines the number of Job Queue Entry events within a Time Period To know more how to troubleshoot JQ, please visit: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/telemetry-job-queue-lifecycle-trace>

jobQueueEventId ↑↓	noOfEvents ↑↓	eventName ↑↓
AL0000E24	4	Enqueued
AL0000E25	4	Started
AL0000E26	4	Finished

Figure 11.32: Layout of workbook with Markdown text

Congratulations! You have now created your first workbook, aka TSG. You could now go back to the logs, create and run other queries, and decide to add (append) them in this workbook or create new ones. In the **Workbook** section in Application Insights, just click on the **Browse across galleries** button and you will find the Azure workbooks just created. These workbooks could be run or even pinned to a new or existing dashboard as a drill-through element.

The last hint on workbooks is related to giving you a good starting-point sample library for the most common troubleshooting experience. This can be found in the following official GitHub repo: [https://github.com/microsoft/BCTech/tree/master/samples/AppInsights/TroubleShootingGuides/D365BC%20Troubleshooting%20Guides%20\(TSG\)/content](https://github.com/microsoft/BCTech/tree/master/samples/AppInsights/TroubleShootingGuides/D365BC%20Troubleshooting%20Guides%20(TSG)/content).

These workbook samples have been developed to target Azure Data Studio and Jupyter notebooks but could easily be adapted to Application Insights. You might think of it as homework to recreate all of them as Azure workbooks in your own subscription.

Now that we have described some of the monitoring features offered by Application Insights, let's jump into other tools that could be used to analyze telemetry data.

## Tools to analyze telemetry data

One of the building blocks of Azure Monitor is that APIs are fully supported. In other words, when signals are ingested into Application Insights tables, these can be accessed externally through standard interfaces. As API requirements, these are fully documented by Microsoft at *Azure Monitor Log Analytics API overview*: <https://learn.microsoft.com/en-us/azure/azure-monitor/logs/api/overview>. Let us give it a try together with the demo database first and then use our own Application Insights resource.

To consume an API, we could use a vast variety of tools; if you are familiar with APIs already, feel free to choose the one that you like or are most familiar with, including *Postman* or even the Dynamics 365 Business Central HttpClient data type. It is up to you. If you are a beginner or simply want to explore another tool that you are not using, let's follow this example using yet another great Visual Studio Code extension, **REST Client**:

1. Open Visual Studio Code in a new window (*Ctrl+Shift+N*).
2. Go to **Extensions** in the action bar.
3. Search for the **REST Client** extension in the marketplace and install it.

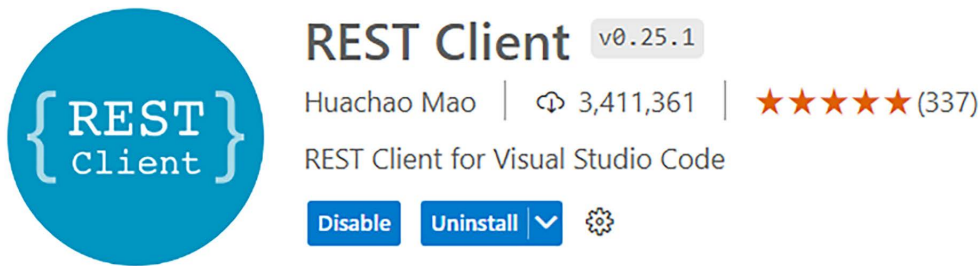


Figure 11.33: REST Client extension

4. Create a new file called, for example, *TelemetryAPI.rest*, and save it.
5. Write the following lines to test the access to the public demo database:

```
POST https://api.loganalytics.azure.com/v1/workspaces/DEMO_WORKSPACE/
query
X-API-Key: DEMO_KEY
Content-Type: application/json

{
  "query": "AzureActivity | summarize count() by Category"
}
```

6. On top of the POST declaration, the extension generates an actionable hyperlink called **Send request**; click it. This will generate a POST request and, whether successful or not, a response should be sent back, like the following:

```
HTTP/1.1 200 OK Date: Thu, 18 May 2023 13:43:30 GMT Content-Type:
application/json; charset=utf-8 Content-Length: 127 Connection: close
```

```
via: 1.1 draft-oms-56d4c9f488-p58f1 X-Content-Type-Options: nosniff
Access-Control-Allow-Origin: * Access-Control-Expose-Headers: Retry-
After, Age, WWW-Authenticate, x-resource-identities, x-ms-status-location
Vary: Accept-Encoding Strict-Transport-Security: max-age=15724800;
includeSubDomains { "tables": [ { "name": "PrimaryResult", "columns": [
{ "name": "Category", "type": "string" }, { "name": "count_", "type":
"long" } ], "rows": [ ] } ] }
```

If a successful response is sent back, then we can proceed and change the syntax to post a query to an Application Insights resource using an API key.



Application Insights also supports OAuth 2.0 authentication with different flows. The flows supported are client credentials, authorization code, and implicit.

7. Go to your Application Insights resource and, from the activity bar, choose **API Access**.



## API Access

Figure 11.34: API Access option

8. Copy the **Application ID** field value.

[+ Create API key](#) [Delete API key](#) [Help](#)

Application ID ⓘ

8dec0de6-8bd1-4e21-97f9-3ab87aac6f7e

Figure 11.35: Application ID field

9. Click on **Create API Key** and provide a name such as, for example, **APIKey**, and allow it to **Read telemetry** by enabling the checkbox.
10. Click **Generate Key**. This will generate an API key, like the one below. Take note of it, since it will be the last time you see it (it is not stored anywhere).

API keys are used by applications outside the browser to access this resource.

Your API keys should be managed like passwords. Keep them secret.



Key:

wswynmyidhgl1c0w6uhxx7c7bzqp6sioajc8c33



**Make sure you copy this key now. We don't store it, and after you close this blade you won't be able to see it again.**

Figure 11.36: API key value

- Now that we have both the application ID and API key, we could send a POST request, using REST Client, to retrieve, for example, all job-queue-related signals in a period that spans the last 10 days (P10D). See below:

```
POST https://api.applicationinsights.io/v1/apps/8dec0de6-8bd1-4e21-97f9-3ab87aac6f7e/query
X-API-Key: wswynmyidhgl1c0w6uhxx7c7bzip6sioajc8c33
Content-Type: application/json

{
  "query": "traces | where customDimensions.aadTenantId
== '72fdeebd-28e2-4685-8494-c1a417aa6d59' and customDimensions.
environmentType == 'Production' and customDimensions.
environmentName == 'Production' | where customDimensions.eventId
in ('AL0000E24', 'AL0000E25', 'AL0000E26', 'AL0000HE7') | extend
jobQueueEventId = tostring(customDimensions.eventId) | summarize count()
by jobQueueEventId | extend noOfEvents = toint(count_), eventName =
case(jobQueueEventId == 'AL0000E24', 'Enqueued', jobQueueEventId ==
'AL0000E25', 'Started', jobQueueEventId == 'AL0000E26', 'Finished',
jobQueueEventId == 'AL0000HE7', 'Failed', 'UNKNOWN') | project-away
count_ | project jobQueueEventId, noOfEvents, eventName | sort by
jobQueueEventId asc",
  "timespan": "P10D"
}
```



8dec0de6-8bd1-4e21-97f9-3ab87aac6f7e is the application ID and qrigylsec2h08ek2p6ppji8oqud37tqonta2uq35 is the API key just generated. Please substitute these values with your own. The same is valid for the AAD tenant ID (72fdeebd-28e2-4685-8494-c1a417aa6d59) and environment name (Production) in the query parameter.

- On top of the POST declaration, click on **Send request**. This will generate a POST request to your Application Insights resource and return the result of the query in JSON format, like the following:

```
HTTP/1.1 200 OK Date: Mon, 29 May 2023 13:05:05 GMT Content-Type:
application/json; charset=utf-8 Content-Length: 284 Connection: close
via: 1.1 draft-oms-755c8c8584-h4k5s X-Content-Type-Options: nosniff
Access-Control-Allow-Origin: * Access-Control-Expose-Headers: Retry-
After, Age, WWW-Authenticate, x-resource-identities, x-ms-status-location
Vary: Accept-Encoding Strict-Transport-Security: max-age=15724800;
includeSubDomains { "tables": [ { "name": "PrimaryResult", "columns": [
{ "name": "jobQueueEventId", "type": "string" }, { "name": "noOfEvents",
"type": "int" }, { "name": "eventName", "type": "string" } ], "rows":
[ [ "AL0000E24", 597, "Enqueued" ], [ "AL0000E25", 597, "Started" ], [
"AL0000E26", 595, "Finished" ], [ "AL0000HE7", 2, "Failed" ] ] ] }
```

Congratulations! You have just extracted aggregated data from your telemetry out from your Application Insights resource. It is possible to automate the data export to whatever third-party storage or even build a custom portal to gather and display selected data in the way you or your customers want. Last but not least, you could also think of using the `AL HttpClient` data type within Dynamics 365 Business Central to get back aggregated signals. In this way, you might create KPIs directly within the application. This is just an idea.

Instead of creating your own dashboard inside or outside Dynamics 365 Business Central, you might also think of using other (free-of-charge) tools available on the market. The best in class are the well-known Power BI telemetry apps provided by Microsoft.

## Power BI telemetry apps

Initially, telemetry was implemented solely for the purpose of reactively troubleshooting cloud issues arising from the application or the platform. Release after release, the product team realized the big potential of such instrumentation to retrieve useful proactive insights that might help not only IT managers but also business stakeholders to make decisions. The exercise of providing insights on the status (pulse) of the ERP is typically called **observability**. Observability is a pillar of technical or processing data-driven decisions and in its concrete form is represented by aggregated data or charts that could provide an immediate snapshot of a specific entity such as, for example, a technical feature or a business process.

After realizing that potential, Microsoft filled the gap between a mere unstructured information collection – the data ingestion – and its restructuring or aggregation for human (or artificial intelligence) analysis. That was, and currently is, the guiding principle of creating and maintaining Power BI telemetry apps. In other words, these apps come geared up with a pre-set template of simple and complex KQL queries that return structured actionable output in most of the Dynamics 365 Business Central technical areas.

Today, using the Power BI telemetry apps should be a must in all modern deployments that would like to have and maintain an overview of how the ERP is performing and drill through each specific business area or simply partner or ISVs that would like to observe how their vertical or horizontal solutions are used.

Apps are officially released free of charge in the AppSource marketplace, and they serve different purposes depending on the telemetry target and analysis: environmental or app-based. The first one is called **Dynamics 365 Business Central Usage Analytics** and provides dashboards, reports, and KPIs with insights related to how the environment and application are used, errors thrown, and overall performance metrics with a broad overview of how the system is used.

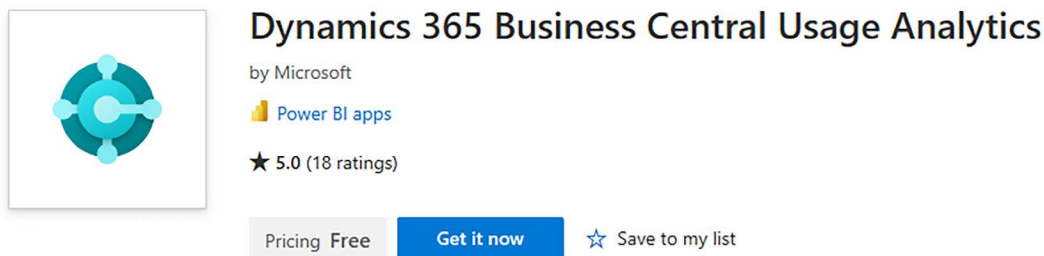


Figure 11.37: Usage Analytics app

The second app, named **Dynamics 365 Business Central App Usage Analytics**, is a spin-off from the first one and it targets how a specific extension, or an ISV application/vertical solution, is used and is performing.

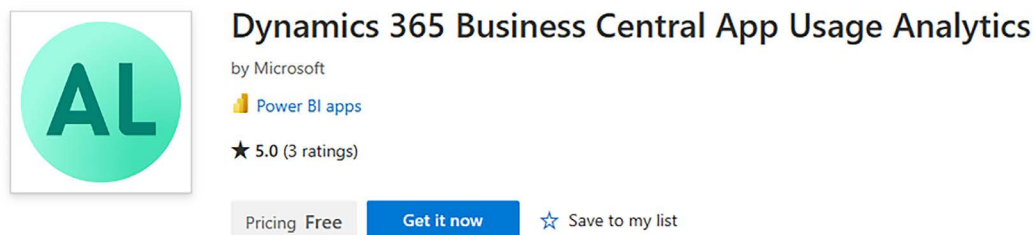


Figure 11.38: App Usage Analytics app

The source code of both apps, in .pbix format, is in the official GitHub repository: <https://github.com/microsoft/BCTech/tree/master/samples/AppInsights/PowerBI/Reports/AppSource>.

Let us see how we could get the app, for example, the environmental one, and connect it to our Application Insights resource.

1. Go to the AppSource Marketplace and in the filters, expand the **Power Platform** tab and check **Power BI apps**.
2. In the **apps** search bar at the top of the page, search for **Dynamics 365 Business Central Usage Analytics**.
3. Click on **Get it**.



This will take you into your Power BI workspace and prompt you to just install the app if you have never done it before, or update only the workspace content without updating the app or update both the workspace content and the app. If you have already installed it for whatever reason, within this example, choose **Install another copy of the app into a new workspace**.

4. Give the workspace a name like **Production Packt** and click **Install**.



The name you choose will be the display name for the app, hence if you need to show the data contained to other stakeholders or IT representatives, it is warmly suggested to use a naming convention that could reflect the business source of the data contained, such as the customer and environment name (if you collect data from a single environment).

5. After the app has been installed, it will contain demo data and you need to provide a connection to your Application Insights resource. Click on **Connect your data**.

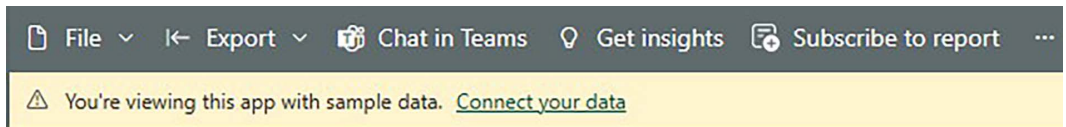


Figure 11.39: Button for connecting to a data source



You might want to use sample data instead of real data to perform the demo or showcase the app usage internally.

6. Fill in the most relevant parameters:
  - **Timezone:** This will be used to transform UTC datetime, date, and time fields (typically timestamps) into the desired time zone.

- **Application Insights application ID:** Retrieved from the API Access tab.

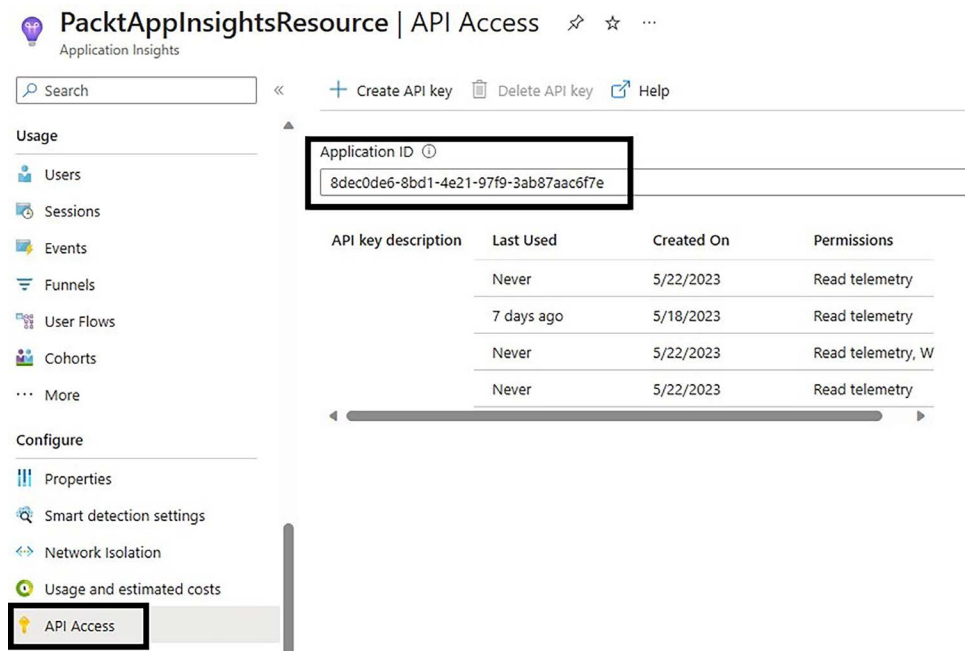


Figure 11.40: Locating an application ID

- **Lookback period:** How far back in time, expressed in days, you would like to retrieve data from the Application Insights resource. You could choose, for example, **30 days**.

Since its advent, Microsoft added more parameters to cover many possible and different scenarios and features. To learn more, please visit *Analyze and Monitor Telemetry with Power BI*: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/telemetry-power-bi-app#configure-an-app-after-initial-setup>.

7. Provide credentials to connect to the Application Insights resource. Once authorized, you should see a banner declaring a dataset refresh, like the following:

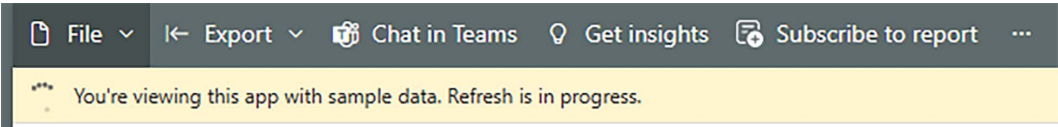


Figure 11.41: Banner indicating a dataset refresh



A refresh might take some time, depending on the volume of data that needs to be downloaded into the Power BI workspace from the Application Insights resource.

8. When finished, your app should look like the one below:

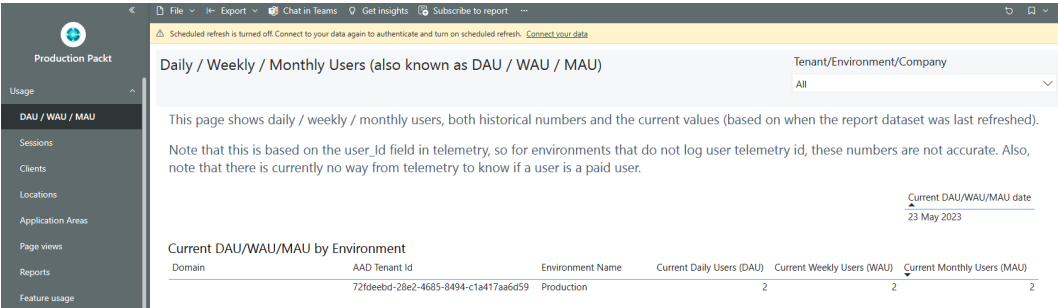


Figure 11.42: Telemetry app connected to a data source

Congratulations! You have just deployed the Power BI telemetry app and connected it to your Application Insights resource to retrieve and analyze data in a Power BI workspace. On the left-side pane, you can click on each session and double-check the values and insights reported. For example, if you drill through the **Errors** tab and select **Job Queue Errors**, you might find the same, or even deeper, analysis we made earlier using KQL queries in the logs table in Azure.

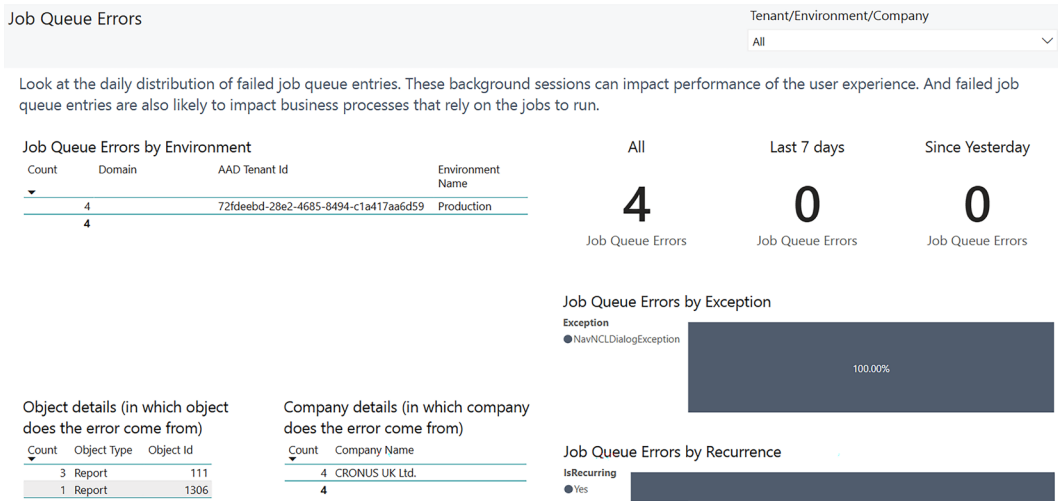


Figure 11.43: Viewing details about job queue errors

To learn more about every single tab and its usage and practical application, we strongly recommend viewing the following videos:

- **20220705 - How to get data driven in your partner practice with telemetry:** <https://www.youtube.com/watch?v=HwmUUOCLVZ8&t=617s>
- **Get Low Friction Go Lives and Optimize Your Investments with Telemetry Data:** <https://www.youtube.com/watch?v=Ka7MWV30yII>

Now that you have gained knowledge about Power BI apps and are sure that your company will adopt this in its daily practice, let's get to the last part of this chapter and instrument an extension with custom application signals to elevate the troubleshooting experience and observability of your AL solutions.

## Custom signals

Telemetry is emitted by the platform and application. Considering application telemetry, we already noticed earlier that this is typically instrumented using code unit 1351 Telemetry Subscribers. For example, signal AL0000E24 is sent when subscribing to OnAfterEnqueueJobQueueEntry published in code unit 453 Job Queue - Enqueue. Below is the relevant code snippet for the subscriber:

```
[EventSubscriber(ObjectType::Codeunit, Codeunit::"Job Queue - Enqueue",
'OnAfterEnqueueJobQueueEntry', '', false, false)]
local procedure SendTraceOnAfterEnqueueJobQueueEntry(var JobQueueEntry: Record
"Job Queue Entry")
var
    TranslationHelper: Codeunit "Translation Helper";
    Dimensions: Dictionary of [Text, Text];
Begin
    TranslationHelper.SetGlobalLanguageToDefault();
    SetJobQueueTelemetryDimensions(JobQueueEntry, Dimensions);

    if IsNullGuid(JobQueueEntry."System Task ID") then
        Telemetry.LogMessage('0000FNY',
            StrSubstNo(JobQueueEntryNotEnqueuedTxt,
                Format(JobQueueEntry.ID, 0, 4)), Verbosity::Warning,
            DataClassification::OrganizationIdentifiableInformation,
            TelemetryScope::All, Dimensions)
    else
        Telemetry.LogMessage('0000E24',
            StrSubstNo(JobQueueEntryEnqueuedAllTxt,
                Format(JobQueueEntry.ID, 0, 4)),
            Verbosity::Normal,
            DataClassification::OrganizationIdentifiableInformation,
            TelemetryScope::All,
            Dimensions);
```

```

    TranslationHelper.RestoreGlobalLanguage();
end;

```

The core part of the code is the AL statement `Telemetry.LogMessage`, which will end up using an AL interface like the following:

```

interface "Telemetry Logger"
{
    procedure LogMessage(EventId: Text; Message: Text; Verbosity: Verbosity;
        DataClassification: DataClassification; TelemetryScope: TelemetryScope;
        CustomDimensions: Dictionary of [Text, Text]);
}

```

It is possible, then, to implement your own telemetry subscriber code unit or fall back to directly using the AL method `Session.LogMessage` (or simply `LogMessage`) inside your solution's code. Let us demonstrate this in a few steps.

1. Log in to your Dynamics 365 Business Central environment.
2. Search for **User Setup** and set **Time Sheet Admin** to **Yes** for the user that is running the job queue entry.
3. Search for **Job Queue Entries** and restart the job queue entry that is running report **111 Customer Top 10 List**.
4. Open the extension **PKT Make A Report Fail** in Visual Studio Code.
5. Change code unit 50139 PKT ReportSubscribers, the function `MakeReportFail`, as reported below:

```

local procedure MakeReportFail()
var
    UserSetup : Record "User Setup";
    TimeSheetAdminErr: Label 'You must be Time Sheet Admin to run this
report.';
    User: Record User;
    Dimensions: Dictionary of [Text, Text];
begin
    If UserSetup.get(UserId) then begin

        User.SetRange("User Name",UserId);
        User.FindFirst();

        Dimensions.Add('UserSecurityId',format(User."User Security ID"));
        Dimensions.Add('IsTimeSheetAdmin',Format(UserSetup."Time Sheet
Admin."));
    end;
end;

```

```

    Session.LogMessage('PKT00001',           //Event id
        'Checking Time Sheet Admin.',       //Message
        Verbosity::Normal,                  //Verbosity
        DataClassification::SystemMetadata, //Data Classification
        TelemetryScope::All,                //Telemetry scope
        Dimensions);                        //Custom Dimensions

    if not UserSetup."Time Sheet Admin." then
        Error(TimeSheetAdminErr);

    end;
end;
}

```

This code will send a signal with ID ALPKT00001, together with all other standard and platform signals, into the Application Insights resource due to the `TelemetryScope::All`.



While the event ID added is PKT00001, the platform is always adding an AL affix by default on all custom signal IDs.

6. Bump the version number of the extension in the `app.json` file (for example, `"version": "1.0.0.1"`).
7. Build the extension and deploy it.

After deploying the extension, just wait for at least 5 minutes to let the job queue run a few times, then go to **User Setup** and remove the tick on the **Time Sheet Admin** column. This will once again generate the error in **Job Queue Entries** (and send yet another alert in your inbox if the alert has not been disabled). Let us check what happened in Azure Application Insights by analyzing the logs with the following KQL query:

```

traces
| where customDimensions.eventID == "ALPKT00001"
| extend isTimeSheetAdmin = tostring(customDimensions.alIsTimeSheetAdmin)
| extend userSecurityId = tostring(customDimensions.alUserSecurityId)
| project timestamp, isTimeSheetAdmin, userSecurityId
| sort by timestamp desc

```


And the result should be like that reported below:

Results		Chart	
timestamp [UTC]		isTimeSheetAd...	userSecurityId
>	5/26/2023, 7:22:05.636 AM	No	{4E56C7A9-2325-4B03-B4A1-B044F7487906}
>	5/26/2023, 7:20:04.689 AM	Yes	{4E56C7A9-2325-4B03-B4A1-B044F7487906}
>	5/26/2023, 7:18:02.339 AM	Yes	{4E56C7A9-2325-4B03-B4A1-B044F7487906}
>	5/26/2023, 7:15:59.526 AM	Yes	{4E56C7A9-2325-4B03-B4A1-B044F7487906}
>	5/26/2023, 7:13:57.885 AM	Yes	{4E56C7A9-2325-4B03-B4A1-B044F7487906}

Figure 11.44: Analyzing logs for a job queue entry

Congratulations! You have created your first custom signal and used it proficiently to find out or corroborate the root cause of a failing job queue entry. Now it is up to you to implement it in your own extensions.

But what if you are an ISV creating AppSource apps to be deployed in a customer environment, or want to keep track of your own signals separately (the ones that belong to your extensions or solutions)? In the manifest file – `app.json` – of any extension, there is a specific parameter, "applicationInsightsConnectionString", that must be used to add the ISV or extension owner Application Insights resource connection string.



To learn more about it, please visit *Dynamics 365 Business Central: adding partner's telemetry on your apps*: <https://demiliani.com/2020/09/01/dynamics-365-business-central-adding-partners-telemetry-on-your-apps/>.

By instrumenting the code with the `TelemetryScope` (`All` or `ExtensionPublisher`) parameter, you could then choose to send the signals to all the ingestion points configured, such as, for example, the one added in the tenant admin center and the one defined in the extension itself, or strictly send signals only to the Application Insights resource specified in the `app.json` file connection string.

In this way, you can analyze signals coming from a specific extension or solution. You could also think of using the free Power BI app that has been developed expressly for this purpose – **Dynamics 365 Business Central App Usage Analytics** – to gain more insights into how your solution is used and how it is performing.

## Summary

In this chapter, we looked at the fundamentals of telemetry, what signals are, and how they have been built in detail. This should be fruitful in understanding which signals or family of signals to use within a specific troubleshooting task.

Then, we practiced setting up an Application Insights resource and connecting a Dynamics 365 Business Central environment to send telemetry traces to it.

We learned how to create KQL queries to aggregate, sort, slice, and dice the telemetry data ingested and exploited the potential of using them as a methodology for troubleshooting and observability tasks. Within this context, we have also gained experience in setting up alerts and creating dashboards to automate the actionability of data analysis.

We left the best for last: Power BI telemetry apps. These are ready-to-use templates to immediately and proficiently start with telemetry within your organization.

Once we finished with the tooling, we learned how to implement our own custom signals and keep track of them.

In the next chapter, we will pack up everything we have learned about coding, debugging, and telemetry and apply development best practices to streamline the performance of our extensions or solutions.

## Leave a review!

*Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.*



*\*Limited Offer*

# 12

## Coding for Performance

When writing extensions for Dynamics 365 Business Central, it's important to go beyond just thinking about the business process and also consider the performance of your code.

There is most likely something you could do to write more efficient AL code and maximize the performance of your extensions. Being aware of the recommended patterns and guidelines will improve your coding and lead to more successful projects. Throughout your career, this will save you time dealing with issues later down the line, and, more importantly, it will help you grow in your role and become a developer that people will learn from and depend on.

This chapter aims to give you some general guidance, and it will cover the following topics:

- Defining an efficient data access layer
- Writing efficient pages and reports
- Writing performant installation and upgrade AL codeunits
- Events and performance
- Running asynchronous patterns
- Testing and validating performance

### Defining an efficient data access layer

When thinking about performance in an ERP solution, the first level that we need to evaluate is the data access layer. Are we creating extensions that access our data efficiently or inefficiently?

When we design a solution, we are often required to create tables and fields. To have the best performance, it's required to also add indexes to tables according to the way your AL code will access the data.

A table object in AL must have a primary key (which uniquely identifies each record in the table) but it can also have many secondary keys (indexes on SQL). The primary key is composed of up to 16 fields in a record. It's technically possible to create a primary key based on up to 20 fields in the development environment, but due to SQL Server limitations, only the first 16 are used.

In the SQL data layer, table extension objects inherit the primary key of the table object they extend (the base table object), and any key that you define in a table extension object is considered a secondary key. A key in a table extension object can include fields from the base table object or the table extension object.

The appropriate keys and the number of active keys to use are always a compromise between maximizing the speed of data retrieval and data updates. Secondary keys help to retrieve data faster but they can slow down writing performance because indexes must be maintained.

Dynamics 365 Business Central also supports the definition of nonclustered columnstore indexes. You can use a non clustered columnstore index to efficiently run real-time operational analytics on the Business Central database. Using a nonclustered columnstore index can improve the performance of analytical queries on large tables. For instance, this can be a neat way of avoiding the locking issues that **SumIndexField Technology (SIFT)** indexes sometimes impose on the system.

In AL, you can define a columnstore index for a table in the following way:

```
ColumnStoreIndex = Field1,Field2;
```

There can be only one nonclustered columnstore index in a table.



For more information about columnstore indexes on SQL, I suggest you check this link: <https://learn.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview>.

When searching for records, AL offers different methods for retrieving data. Dynamics 365 Business Central records are retrieved using **multiple active result sets (MARS)**. The available AL methods to retrieve data are the following:

- `Record.Get` gets a single record based on primary key values.
- `Record.Find` gets a single record based on the primary keys in the record and any filter or range that has been set. You're unlikely to ever use `Find` on its own as a keyword but, as we will see, it's a building block for more useful methods.
- `Record.Find(' - ')` and `Record.Find(' + ')` are optimized for reading primarily from a single table when the application might not read all records. `Find(' - ')` is implemented by issuing a self-tuning TOP X call, where X can change over time, based on the statistics of the number of rows read. You would use `Find(' - ')` and `Find(' + ')` only when you are accessing very large amounts of data. In the vast majority of cases, filtered record sets are small enough to use `FindSet()`.
- `Record.FindSet(ForUpdate)` is optimized for reading the complete set of records in the specified filter and range. `FindSet` is not implemented by issuing a TOP X call. This should be the default for retrieving multiple records.
- `Record.FindFirst` and `Record.FindLast` are optimized for finding the single first or last record in the specified filter and range.

- `Record.IsEmpty()` is used when you just need to know whether a filtered record set returns anything.

When you need to retrieve data and calculate the values of some `FlowField`, you can use the `SetAutoCalcFields` method to avoid extra loops. `SetAutoCalcFields` automatically updates the `SELECT` query on SQL with joins for each `FlowField` you need to calculate, and this results in one query with all your `FlowFields` calculated. This capability was created to avoid having to run `CalcField` in code and to provide a way to join tables in the initial query. So, we are avoiding extra loops by overcoming the need to use `CalcFields` inside an `Item` repeat loop. SQL Server is very good at joining tables, and this technique takes advantage of that.

An example of usage is the following:

```
local procedure CheckAvailability()
    var
        Item: Record Item;
    begin
        Item.SetRange(Type, Item.Type::Inventory);
        Item.SetAutoCalcFields(Inventory, "Qty. on Purch. Order", "Qty. on
Sales Order");
        if Item.FindSet() then
            repeat
                if Item.Inventory < Item."Qty. on Sales Order" then
                    //do some actions
                until Item.Next() = 0;
            end;
        end;
```

In this example, the initial `Item` query joins the **Item Ledger Entry (ILE)** table, which SQL Server can do very quickly. When you leave out `AutoCalc` and you run `CalcFields` for each `Item`, it has to issue ILE queries for each item.

Bulk operations can help improve performance in code. Using `ModifyAll` and `DeleteAll` can improve performance by limiting the amount of SQL calls needed. However, be aware that `ModifyAll` and `DeleteAll` will revert to individual calls if any of the following conditions exist:

- There is trigger code in the table.
- There are event subscribers to the following events: `OnBeforeModify`, `OnAfterModify`, `OnGlobalModify`, `OnBeforeDelete`, `OnAfterDelete`, `OnGlobalDelete`, and `OnDatabaseModify`.
- Security filtering is active.
- The table contains `Media` or `MediaSet` data type fields.
- There are fields that are added through companion tables.

When designing AL solutions with performance in mind, it's extremely important to think about table extensions. Having a long chain of `tableextension` objects of the same heavily used table affects performance a lot (more information can be found here: <https://demiliani.com/2020/12/28/dynamics-365-business-central-the-impact-of-tableextensions/>).

When you have a table extension, every data read operation with `Find` or `FindSet` is translated into a SQL query (the `SELECT *` operation is the default operation performed) with joins between the main table and every table extension of that table, resulting in a performance degradation the more table extensions are involved. The problem is not necessarily that there is a lot of data but, rather, that the data structures may be poorly designed or indexed inefficiently.

To avoid those extra joins if not needed, AL has a method called `SetLoadFields` that you should always use to optimize performances:

```
[Ok := ] Record.SetLoadFields([Fields: Any,...])
```

`SetLoadFields` sets the fields to be initially loaded when the record is retrieved from its data source. The default query for any table in Business Central generates a `SELECT *` query that includes all fields from all table extensions. By using `SetLoadFields`, the query will only read from the table and table extensions where those fields are defined. This is called *partial record loading*.

An example of usage is the following:

```
procedure CalculateAverageStdCost(): Decimal;
var
    Item: Record Item;
    SumTotal: Decimal;
    Counter: Integer;
begin
    Item.SetLoadFields(Item."Standard Cost");
    if Item.FindSet() then begin
        repeat
            SumTotal += Item."Standard Cost";
            Counter += 1;
        until Item.Next() = 0;
        exit(SumTotal / Counter);
    end;
end;
```

Here, only the `Standard Cost` field of the `Item` table is retrieved, avoiding extra joins and loading extra fields.

When a record is loaded as a partial record and you try to access a field not previously loaded, the platform executes a just-in-time loading of that field by doing an implicit `GET` operation on the record to retrieve the missing field. In this case data access is required again, but this time, it's performed via a `GET` operation (primary key), so it's faster.

The Dynamics 365 Business Central platform automatically applies the partial record feature to the following AL objects based on their metadata:

- Reports
- List and ListPart pages

- OData pages
- Table relationship-based lookups

Please remember that the `OnFindRecord` and `OnNextRecord` pages trigger conflict with the partial record feature with `List` and `ListPart` pages, so if these triggers are defined in the metadata, the partial record feature won't be applied.

## Table extension changes from Dynamics 365 Business Central version 23

The Dynamics 365 Business Central 2023 Wave 2 release (version 23) introduces an important change in how table extensions are handled at the database layer.

In previous releases, when a developer extended a table, the fields from the table extension were stored in a separate table (companion table) in the database.

Starting from this release, all the newly added fields from all extensions to a table are now stored in the same companion table. In this new model, the server will never need to do more than a single join of the base table to its companion table.

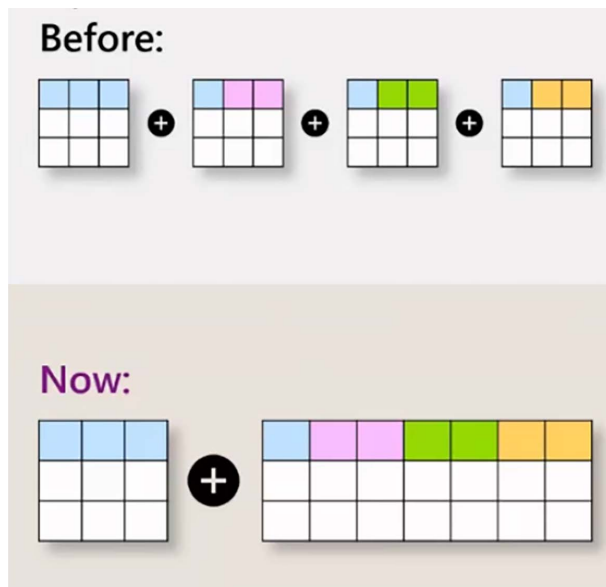


Figure 12.1: Companion table model for Business Central 2023 Wave 2

In the new data structure, only one companion table (named `tablename$ext`) that holds all newly added fields to the base table is created when you extend a table from the AL code. In the new single companion table, all field names contain the app ID that defines the field.

This change does not impact the table model as seen from AL, so no code changes are needed from extension/app publishers. Developers can now organize code with table extensions the way they want to, without having to think about the performance impact of where table extensions are located in apps.

## Setting the transaction isolation level in AL code

To maximize performance in certain scenarios and avoid extra locks, you can also control the SQL transaction isolation level via AL code.

By default, the Dynamics 365 Business Central runtime automatically determines the isolation levels used when querying the database. In AL, you can explicitly control the database isolation level on individual reads on a record instance by using the `ReadIsolation` property in the following way:

```
Rec.ReadIsolation := IsolationLevel::<enum value>
```

Here, `IsolationLevel` can have the following values:

Value	Description
Default	Follows the table's isolation level for reads; same behavior as not setting an isolation level.
ReadCommitted	Allows reads on committed data only, not data that's been modified by other transactions but not yet committed.
ReadUncommitted	Allows the record to read data that has been modified by other transactions but not yet committed (also called dirty reads). A <code>ReadUncommitted</code> transaction takes no locks and ignores locks from other transactions.
RepeatableRead	Ensures that reads stay stable for the life of the current transaction. Until the current transaction completes, the record can't read data that has been modified but not yet committed by other transactions, and other transactions can't modify data that has been read by the current transaction.
UpdLock	Ensures that reads stay consistent for the life of the current transaction. Until the current transaction completes, the record can't read data that has been modified but not yet committed by other transactions, and other transactions with the same isolation level can't read data that was read by the record.

Table 12.1: *IsolationLevel* values

Reads are `ReadUncommitted` when no writes have been done to the table in the current transaction. If a row has been written to the table (or `LockTable` is called) in the current transaction, reads are done with `UpdLock` against the table. `ReadCommitted` is the new default isolation level from Dynamics 365 Business Central version 23.



`LockTable` does not “lock a table.” As we can see in `UpdLock`, what we are doing is setting the isolation level of the query.

Some examples of the usage of this property are the following:

```
// Gets the next "Entry No." and Locks just last row.
// Without causing the rest of transaction to begin taking locks.
local procedure GetNextEntryNo(): Integer
var
    GLEntry: Record "G/L Entry";
begin
    GLEntry.ReadIsolation := IsolationLevel::UpdLock;
    GLEntry.FindLast();
    exit(GLEntry."Entry No." + 1)
end;

// Do a count of records
// Only count records that have been committed
local procedure CountsWithoutLocking(): Integer
var
    GLEntry: Record "G/L Entry";
begin
    GLEntry.ReadIsolation := IsolationLevel::ReadCommitted;
    exit(GLEntry.Count());
    // even when current session is in write mode on "G/L Entry"
    // then this count operation does not take any locks
end;
```

When designing complex processes in AL code, consider handling transactions carefully. Remember that in a cloud environment, your application is not alone, and third-party extensions could affect your transactions (commits).

Let's look at a simple, abstract example to demonstrate this new isolation level:

```
codeunit 50100 MyBusinessProcessMgt
{
    procedure MyBusinessProcess()
    begin
        ExecuteProcess1();
        ExecuteProcess2();
        OnBeforeFinalProcess();
        if ExecuteFinalProcess() then
            Commit();
    end;

    local procedure ExecuteProcess1()
    begin
```

```

    //Your code here...
end;

local procedure ExecuteProcess2()
begin
    //Your code here...
end;

local procedure ExecuteFinalProcess(): Boolean
begin
    //Your code here...
    exit(true);
end;

[IntegrationEvent(false, false)]
local procedure OnBeforeFinalProcess()
begin
end;
}

```

Here, a business process is composed of three different steps. Only if the last step (in this case, `ExecuteFinalProcess`) is successfully executed is the transaction committed. The process raises an integration event (`OnBeforeFinalProcess`) immediately before the last step.

Now, imagine having a third-party extension that does the following:

```

[EventSubscriber(ObjectType::Codeunit, Codeunit::MyBusinessProcessMgt,
OnBeforeFinalProcess, '', false, false)]
local procedure ThirdPartyHandlerAction()
begin
    if ThirdPartyProcess() then
        Commit();
end;

local procedure ThirdPartyProcess(): Boolean
begin
    //Your code here...
    exit(true);
end;

```

This extension subscribes to the `OnBeforeFinalProcess` event, executes a custom process, and then performs an explicit commit on the transaction.

It breaks your business process because the `commit` is executed before the `ExecuteFinalProcess` step.

To avoid things like this, in AL, you can specify the behavior of a commit operation inside a method scope (procedure) or an event by using the `CommitBehavior` attribute, which can have the following values:

- `CommitBehavior::Ignore`: All commit calls are ignored until the method scope ends.
- `CommitBehavior::Error`: This throws an error if a commit operation is executed before the end of the scope of the method.

Here is some example code using this attribute:

```
[CommitBehavior(CommitBehavior::Ignore)]
[IntegrationEvent(false, false)]
local procedure OnBeforeFinalProcess()
begin
end;
```

If the parent method is executed with a more restrictive behavior (for example, `CommitBehavior::Error`), then the current method will continue to respect the more restrictive behavior defined.

## Writing efficient pages and reports

Having responsive and fast-opening pages is essential for giving your ERP users a good user experience.

When designing pages in AL, please always avoid putting too many visible fields on a single page and try to avoid unnecessary recalculations – less is more. Also, try to avoid having too many calculated fields visible on a page, especially if those fields do calculations on large tables. Please remember that setting the field's `Enabled` or `Visible` properties to `false` isn't enough - fields are included in the query even if you set a field to **Invisible**. Instead, it's better to remove the field definitions from the page or page extension.

To have a more responsive UI when you need calculations on a particular page, avoid heavy operations in the `OnAfterGetRecord` trigger and consider using asynchronous programming and page background tasks. These concepts are explained in *Chapter 6, Advanced AL Development*.

When designing reports, use the partial records feature for your datasets (`SetLoadFields`) and consider using AL query objects to optimize the data that is read from the database.

Dynamics 365 Business Central SaaS natively supports a read-only database replica (the built-in `Read Scale-Out` feature of Azure SQL Database), and you can use that replica for reporting purposes (read-only transactions). In this way, you can reduce the load on the primary database and the impact of complex analysis (locks, etc.) on your users.

When you create a report object (but also an API page or a query), you can use the `DataAccessIntent` property to specify whether the object will be executed on the read-only replica or not:

```
DataAccessIntent = ReadOnly|ReadWrite;
```

From the client, the property value can be overwritten by using **9880 - Database Access Intent List**.

Query objects enable you to retrieve records from one or more tables and then combine the data into rows and columns in a single dataset. Query objects can also perform calculations on data, such as finding the sum or average of all values in a column of the dataset.

There are two types of query objects:

- **Normal:** Can be used to display data in the UI
- **API:** Used to generate web service endpoints (*QueryType = API*)

Query objects have the following advantages:

- Not all fields are read from the database (but only the fields you need are retrieved)
- Fast read performances
- You can join multiple tables

The main disadvantages of using query objects are the following:

- No caching, since the data is always read from SQL.
- No pages built on top of queries. This is a limitation of any page type object, which also applies to query objects.
- Query result sets are not guaranteed to be dynamic. This means that if you insert or modify data in a result set row that you have not yet looped through, then it is not guaranteed that the query result set will include those changes.

Keep in mind that, sometimes, it is faster not to use query objects, since they will always go to the database.

## Writing performant installation and upgrade AL codeunits

To support bulk data transfer between tables, a new data type called `DataTransfer` can be used in AL code.

The `DataTransfer` data type operates on sets instead of operating record by record, and this permits you to create efficient data manipulation logic. `DataTransfer` code runs much faster than code that simply loops through records because the type was specifically designed to provide a fast way to transfer data. `DataTransfer` primarily has the following uses:

- Copying data from one or more fields in a table to fields in another table (for example, when you make some fields obsolete between versions of your extensions)
- Copying data from entire rows in a table to rows in another table (for example, when you make a table obsolete or when you refactor the data structure of your solution)

The `DataTransfer` data type works in `Upgrade` and (from version 22) in `Install` codeunits and no events are raised when using it.

When using the `DataTransfer` data type, you need to follow this pattern:

1. Specify the source and destination tables by using the `SetTables` method.
2. Specify which fields to transfer by calling the `AddFieldValue` or setting a constant value for fields in the destination using `AddConstantValue`, respectively.
3. Define the relationship between the source and destination tables by calling `AddJoin`.
4. Add constraints on the data to transfer by calling `AddSourceFilter`.
5. Invoke the query for transferring data by calling `CopyFields` or `CopyRows`.

An example of usage is the following:

```
local procedure CopyRows()
var
    dt: DataTransfer;
    to: Record ToTable;
begin
    dt.SetTables(Database::FromTable, Database::ToTable);
    dt.AddFieldValue(1, to.FieldNo("id"));
    dt.AddFieldValue(2, to.FieldNo("SmallCodeField"));
    dt.AddFieldValue(3, to.FieldNo("IntField"));
    dt.CopyRows();
end;
```

Note that `AddFieldValue` 1, 2, and 3 refer to field numbers.

In the following example, consider having `Source` and `Destination` tables. We want to make field `S3` obsolete in the `Source` table and then copy field `S3` into field `D3` of the `Destination` table in rows where field `S2` is equal to `A`.

The code to do that is the following:

```
local procedure CopyFields()
var
    dt: DataTransfer;
    dest: Record Destination;
    src: Record Source;
begin
    dt.SetTables(Database::Source, Database::Destination);
    dt.AddFieldValue(src.FieldNo("S3"), dest.FieldNo("D3"));
    dt.AddSourceFilter(src.FieldNo("S2"), '=%1', 'A');
    dt.AddJoin(src.FieldNo("PK"), dest.FieldNo("PK"));
    dt.CopyFields();
end;
```

## Events and performance

When creating AL extensions, it's extremely important to think about the extensibility of your solution. In *Chapter 3, Extension Development Fundamentals*, and *Chapter 4, Developing a Customized Solution for Dynamics 365 Business Central*, we saw that we need to use events to write an extensible solution. But can events impact performance?

In general, a publisher (who raises the event) has no impact on performance.

Event subscribers could instead affect performance.

If you write logic attached to a table event (OnInsert, OnModify, and so on) remember that the runtime will issue SQL update/delete statement rows in a for loop rather than in a single SQL statement. As said before, remember also that bulk operations are impacted if you have events subscribing to the table events, and these operations are converted to normal single SQL operations.

As a best practice, you should handle events in a single instance codeunit if possible, and the size of the subscriber should be as small as possible.

## Running asynchronous patterns

When you have complex calculations that need to start from a UI trigger (like AL code that needs to update statistical values on a page) and these calculations require time to complete, to avoid blocking the UI thread, it's often recommended to offload the task execution to a background session.

To execute a task in the background, you can use the following:

- **StartSession:** Permits you to execute a new session in parallel. The new session will run in the background and can perform read and write transactions.
- **Task Scheduler:** Used in code to execute tasks in the background. These tasks are queued up and survive between server restarts (the process will be retried when the server is up again).
- **Page Background Task:** Permits you to execute read-only tasks in a background session that can be started from a main UI thread. Details about how to use them were explained in *Chapter 5, Writing Code for Extensibility*.

## Using StartSession in AL code

To use StartSession in AL code, you need to:

1. Create a codeunit with the code you want to execute in the background session.
2. Start the codeunit with the StartSession AL method. You can find more information about this method syntax at the following link: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/session/session-startsession-integer-integer-string-table-duration-method>.

Imagine that we want to create a process for setting all the Customer records in the Blocked field to blank and then doing other things (a long-running task). We can create a codeunit object defining the task steps:

```

codeunit 50100 PACKTBackgroundProcess
{
    trigger OnRun()
    begin
        UnblockCustomers();
        //Other processes here...
    end;

    local procedure UnblockCustomers()
    var
        Customer: Record Customer;
    begin
        if Customer.FindSet() then
            repeat
                Customer.Validate(Blocked, Customer.Blocked::" ");
                Customer.Modify();
                //Do other things here...
            until Customer.Next() = 0;
        end;
    end;
}

```

Then, on the **Customer List** page, we can add an action to start the task in a background session by using `StartSession`:

```

pageextension 50100 PACKTCustomerListExt extends "Customer List"
{
    actions
    {
        addlast(processing)
        {
            action(PACKTUnblockCustomersSS)
            {
                Caption = 'Unblock all Customers via STARTSESSION';
                ApplicationArea = All;

                trigger OnAction()
                var
                    SessionID: Integer;
                    result: Boolean;
                    SuccessMsg: Label 'Customers unblocked successfully.';
                    ErrorMsg: Label 'Error unblocking customers.';
                begin

```

```

        result := StartSession(SessionID,
Codeunit::PACKTBackgroundProcess);
        if result then
            Message(SuccessMsg)
        else
            Error(ErrorMsg);
        end;
    }
}
}
}
}

```

When using `StartSession`, please remember the following aspects:

- Each background session has the same impact on system resources as a regular user session. Don't use it for frequently running tasks; it's better to use it for heavy tasks that don't run frequently.
- You can debug a background session only in on-premises environments; on SaaS, debugging a background session is not supported.

## Using Task Scheduler in AL code

To schedule the previously created task by using Task Scheduler from AL code, you need to use the `TaskScheduler` data type, as in the following code:

```

action(PACKTUnblockCustomersTS)
{
    Caption = 'Unblock all Customers via TASK SCHEDULER';
    ApplicationArea = All;

    trigger OnAction()
    begin
        TaskScheduler.CreateTask(Codeunit::PACKTBackgroundProcess, 0, true, Rec.
CurrentCompany, CreateDateTime(Today + 1, 0T));
    end;
}

```

Here, the `TaskScheduler.CreateTask` method is used to schedule a codeunit at a specified time. Details about the `TaskScheduler` data type can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/taskscheduler/taskscheduler-data-type>.

In the `CreateTask` method, we need to pass the codeunit to schedule, the ID of a failure codeunit (0 if you don't have a codeunit to start if the task fails), the task status to set (true if you want to set the task as ready), the Business Central company where the task must run, and the time when the task will be scheduled for execution.

In Dynamics 365 Business Central, you can see all the scheduled tasks via the **Scheduled Task** page. Here, you will see all the tasks scheduled via code or a job queue.

The TaskScheduler object permits you to automatically retry tasks if an execution fails. More details about the retry cycle can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-task-scheduler#retrycycle>.

## Testing and validating performances

Before deploying a Dynamics 365 Business Central solution into a production environment, you should do some performance testing or measures.

In AL code, you can use the SessionInformation data type to measure the number of SQL statements or rows read that your code is doing. For example, take the following AL code:

```
SqlRowsRead := SessionInformation.SqlRowsRead();  
SqlStatementsExecuted := SessionInformation.SqlStatementsExecuted();
```

This gives information about the number of SQL statements executed and the number of SQL rows read in a session.

More information about the SessionInformation data type can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/sessioninformation/sessioninformation-data-type>.

To test code performance, it's also extremely important to use telemetry. The code-related telemetry signals to check are under the following families of signals:

- Database locks and deadlocks
- Long-running AL methods
- Long-running SQL queries
- Sessions started
- Page views
- Reports
- Job queue tasks

When investigating AL code performance, other tools are recommended:

- In-client performance profiles
- The AL performance profiler provided by Visual Studio Code

These two tools (explained in *Chapter 10, Debugging*) are useful to simulate a business process and then record performance metrics (the .alcpuprofile file), which can then be analyzed.

The last tool that we recommend using is the **Performance Toolkit** extension. This toolkit is useful if you want to simulate workloads and monitor performance regressions.

This tool is composed of a set of components:

- **Performance Toolkit app** (installable on your Dynamics 365 Business Central environment via the AppSource marketplace)
- **Performance Toolkit extension for Visual Studio Code** (installable via the Visual Studio Code Marketplace); useful to set up a new performance toolkit project and PowerShell scripts for testing
- **PowerShell scripts and sample test cases**

The recommended way to use the **Business Central Performance Toolkit (BCPT)** is by using the Visual Studio Code extension. When installed, you can start a new Performance Toolkit project by using the following command:

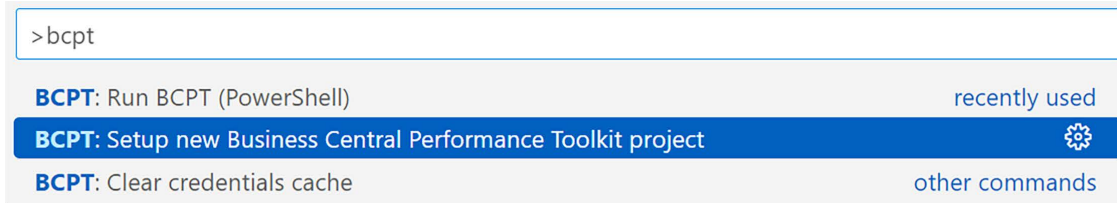


Figure 12.2: Command for a new BCPT project

Then, you can select your Dynamics 365 Business Central environment type (**SaaS** or **Docker**):

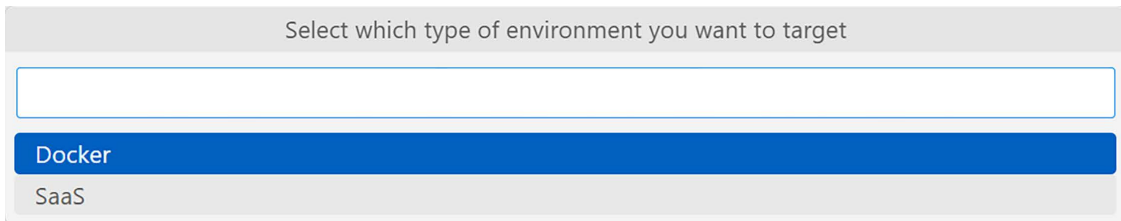


Figure 12.3: Selecting an environment type

When selecting the environment to target, the extension verifies that all the prerequisites are installed into the target environment (if not, it will install them) and then creates a new BCPT project, as follows:

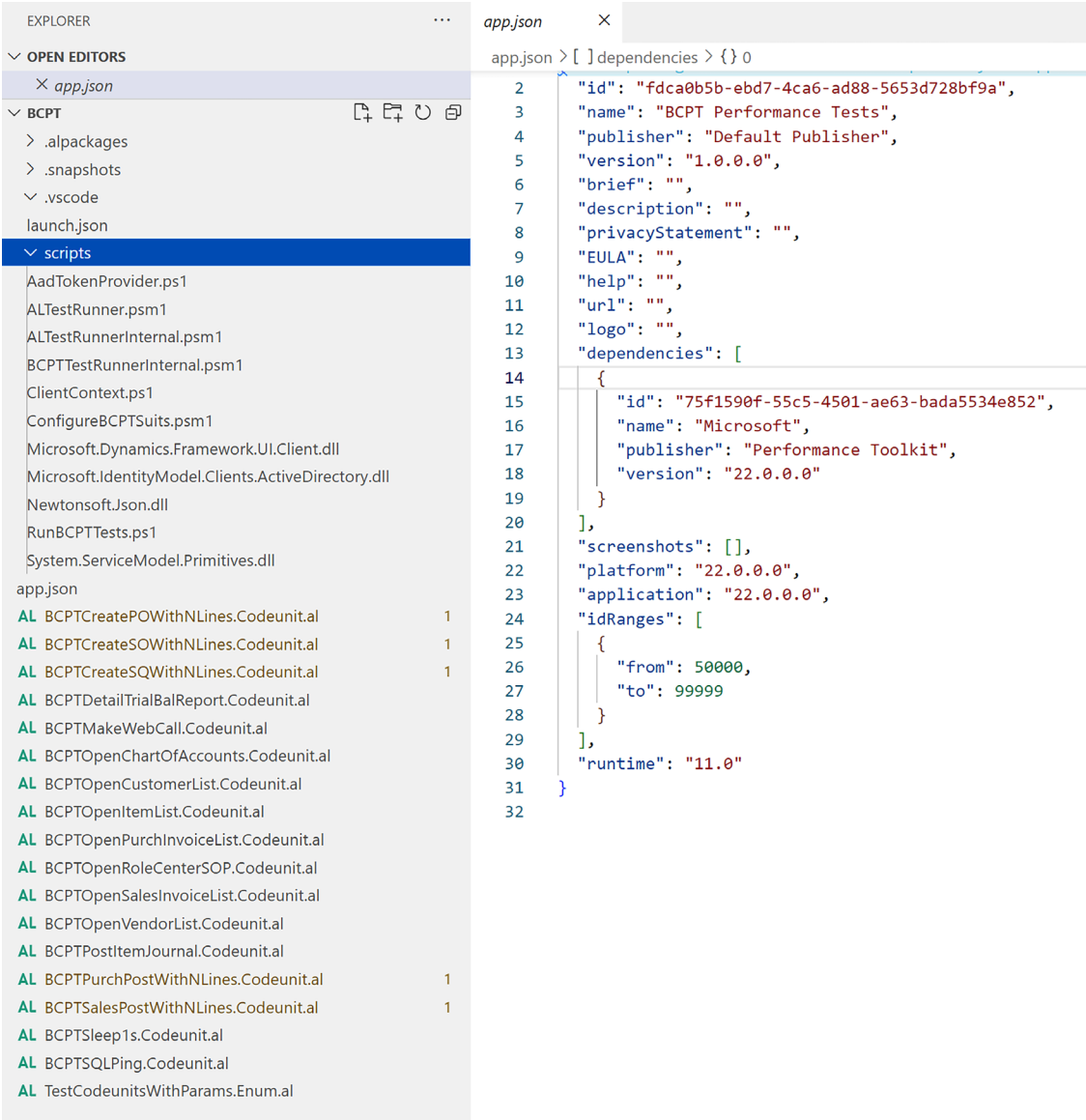


Figure 12.4: Project created in target environment

As you can see, in the project, there are:

- PowerShell scripts to start the tests
- AL objects that define the tests

The test scenarios that are provided by default in the project can then be extended according to your needs. You can also create your own test scenarios.

You can deploy these sets of tests in your Dynamics 365 Business Central environment (as a custom extension) and then, from Business Central, you can create a new **BCPT Suite** targeting your tests:

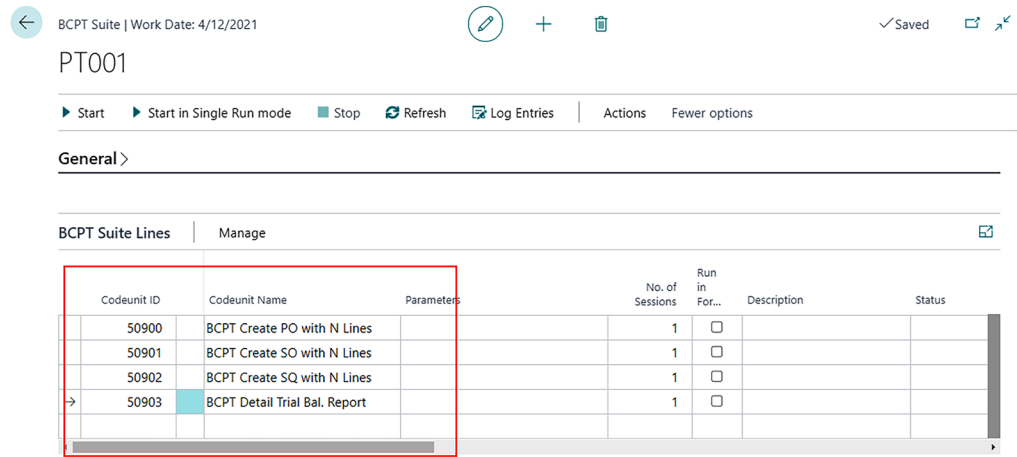


Figure 12.5: Creating a BCPT Suite to manage tests

On the BCPT Suite page, the **Default Min. User Delay** and **Default Max. User Delay** fields in the header permit you to simulate delays between actions when executing your tests.

More information about this toolkit can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-performance-toolkit>.

## Summary

In this chapter, we saw some tips to remember when writing code with performance in mind. The chapter covered topics related to defining efficient code for accessing data, writing efficient pages and report datasets, events and their impact on performance, asynchronous tasks, and testing performance.

These guidelines are useful for optimizing solutions for performance and they should help you write efficient code.

In the next chapter, we'll see how to use APIs in Dynamics 365 Business Central.

## Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.



\*Limited Offer

# 13

## Dynamics 365 Business Central APIs

In the previous chapter, we saw tips that you can use during development to improve the performance of your AL code.

When working in a Dynamics 365 Business Central project, integration with external systems is often a requirement. Integrations between Dynamics 365 Business Central and other systems and services can be created in different ways. In this chapter, we'll see how to integrate Dynamics 365 Business Central with external applications by using the RESTful APIs exposed by the platform (APIs that conform to the standard REST guidelines), and the focus will be on the following topics:

- Using the OData protocol for APIs
- Handling OAuth 2.0 authentication with Dynamics 365 Business Central APIs
- Using the Dynamics 365 Business Central standard APIs
- Creating custom APIs with Dynamics 365 Business Central for new and existing entities
- Creating applications that use Dynamics 365 Business Central APIs
- Using OData bound and unbound actions
- Using batches for sending API requests in bulk
- Using Dynamics 365 Business Central webhooks
- Using Automation APIs in Dynamics 365 Business Central

By the end of this chapter, you will be able to create RESTful APIs for Dynamics 365 Business Central and use them efficiently for integrations with external applications like third-party systems or services, other APIs, and so on.

## Using the OData protocol for APIs

Every client that can make HTTP calls can consume RESTful APIs. By using the GET, POST, PATCH, and DELETE verbs of the HTTP protocol, you can **Create, Read, Update, and Delete (CRUD)** entities. To make integrations with Dynamics 365 Business Central, the OData protocol and RESTful APIs are the recommended tools to work with.

The **Open Data (OData)** protocol is a web protocol that permits you to perform CRUD operations on tabular data with HTTP calls by using **URIs** (short for **Uniform Resource Identifiers**) for resource identification. These URIs, like URLs, serve as OData **endpoints** that identify resources on the web.

At the moment, there is a simple way to expose an object (like a page, a codeunit, or a query object) as an OData endpoint in Dynamics 365 Business Central. First, you open the **Web Services** page, and then you insert a new record by setting the **Object Type** and **Object ID** fields, give it a **Service Name**, and set the **Publish** field to true.

As you can see in the following screenshot, Dynamics 365 Business Central automatically assigns an OData V4 URL to the published entity, and then you can use this published entity as an OData endpoint by performing HTTP REST calls to the provided endpoints:

**Dynamics 365 Business Central**

Web Services

× If you want to set up an OData connection, for performance and stability reasons consider using an API page instead. Don't show again | API documentation

Search + New Edit List Delete Reload Download Metadata Document More options

Object Type ↑	Object ID	Object Name	Service Name ↑	All Tena...	Publi...	OData V4 URL
Page	5493	cashFlowStatement	ExcelTemplateCashFlowStatement	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	5503	incomeStatement	ExcelTemplateIncomeStatement	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	5497	retainedEarningsStatement	ExcelTemplateRetainedEarnings	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	5502	trialBalance	ExcelTemplateTrialBalance	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	1320	ExcelTemplateCompanyInfo	ExcelTemplateViewCompanyInform...	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	2200	Sales Invoice Document API	InvoiceDocument	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	2201	Sales Invoice Reminder API	InvoiceReminder	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
→ Page	89	Jobs	Job List	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	1007	Job Planning Lines	Job Planning Lines	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	1002	Job Task Lines	Job Task Lines	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...
Page	16	Chart of Accounts	Piano dei conti	<input type="checkbox"/>	<input checked="" type="checkbox"/>	https://api.businesscentral.dynami...

Figure 13.1: OData V4 URL for an entity

When calling an OData endpoint, you can apply filters, use grouping, use FlowFilters, and call business logic by using bound actions (we'll discuss them later in this chapter, in the *Using bound actions* section).

While the Web Services page is currently a simple option for us to get started with using the OData protocol, it is important to know that the Web Services page will be deprecated in the future. Once you're comfortable with understanding how to use HTTP REST calls to OData endpoints, you should move on to learning how to use **dedicated API pages**.

These API pages, although slightly more complex, use the same OData protocol for APIs. Learning how to use them will help you avoid deprecation issues, and also allow you to enjoy some performance and stability advantages, which we will now explore.

APIs in Dynamics 365 Business Central use the same OData protocol under the hood, but they have three main advantages when we talk about integration:

- They have **versioning**, which doesn't exist for web services. This is one of the most important things when doing service integration because you need a stable contract. If you have made improvements to an API, you should keep in mind that changing APIs is breaking and should always be done changing the version.
- They are **webhook supported** (you can publish your API page and then call `/api/microsoft/runtime/beta/webhookSupportedResources?$filter=resource eq 'v2.0*'` to verify that the entity is supported by webhooks).
- They have **namespaces** so you can isolate and group your APIs according to their scope or functional area: `{{shortUrl}}/api/APIPublisher/APIGroup/v1.0/customers('01121212')`.



For more information on how to use RESTful APIs in general, I recommend the following link: <https://www.odata.org/getting-started/basic-tutorial/>.

With these advantages in mind, we should now understand that API pages are the most performant solution when creating OData endpoints. The rest of this chapter will proceed using API pages.

## Configuring OAuth authentication for Dynamics 365 Business Central APIs

To start using Dynamics 365 Business Central APIs, you first need to be authenticated. Dynamics 365 Business Central SaaS requires that you use the **OAuth** protocol for authentication.

**OAuth** is an open standard authorization protocol for securing access to third-party services. OAuth never shares passwords between the actors of a process but instead, it sends authorization tokens, and this token is used to prove the authenticity of the identity.

OAuth permits users to sign in to Business Central APIs using their Microsoft 365 or Microsoft Entra ID credentials in an interactive or non-interactive way. Please note that in Dynamics 365 Business Central on-premises, you can still use basic authentication (username and web service access key).

In this section, we'll see how to configure Microsoft Entra ID (previously named Azure Active Directory or Azure AD) for using **Service-to-Service (S2S)** authentication with Dynamics 365 Business Central APIs. S2S authentication is often the best choice for scenarios where integrations need to run without any user interaction. This will be almost every scenario you face when creating integrations in Business Central.

S2S authentication uses the Client Credentials OAuth 2.0 Flow, which enables you to access resources by using the identity of an application. More information about this authentication flow can be found at the following link: <https://learn.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-client-creds-grant-flow>.

To configure Microsoft Entra and Dynamics 365 Business Central APIs to use OAuth 2.0 with S2S authentication, we need to:

1. Register an application in Microsoft Entra ID.
2. Set the application permissions.
3. Create a client secret.
4. Register a Microsoft Entra ID application in Dynamics 365 Business Central.

## Registering an application in Microsoft Entra ID

From the Azure portal, open **Microsoft Entra ID** and select **App registrations** | **New registration**:

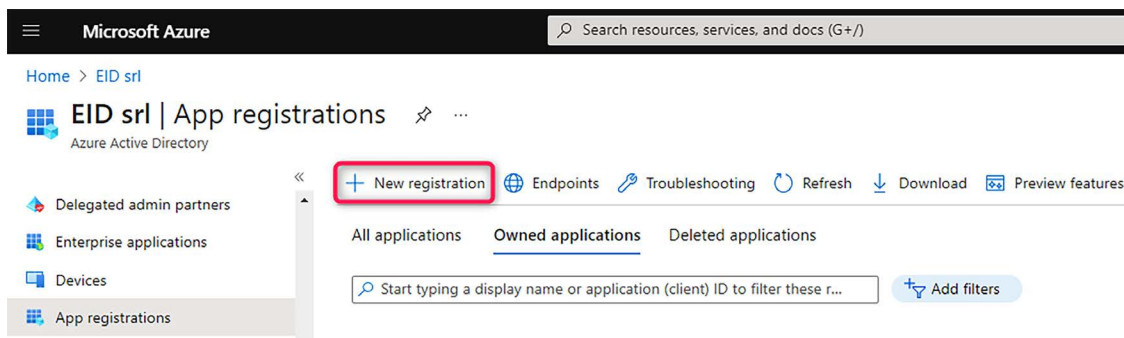


Figure 13.2: App registrations page

Select a name for the app registration and then select **Accounts in this organizational directory only (single tenant)** or **Accounts in any organizational directory (Any Microsoft Entra directory - Multitenant)**, according to your needs. Normally, when you select **Accounts in this organizational directory only (single tenant)**, you allow only users and guests from the tenant where the developer has registered their app. This option is the most common for Line of Business (LOB) applications. When you select **Accounts in any organizational directory (Any Microsoft Entra directory - Multitenant)**, you allow any user from any Microsoft Entra directory to sign in to your multi-tenant application.

Set the **Redirect URI** field with the following value (case-sensitive): <https://businesscentral.dynamics.com/OAuthLanding.htm>.

Then, click on **Register**:

Home > | App registrations >

Register an application ...

\* Name

The user-facing display name for this application (this can be changed later).

PACKTD365BCAPI ✓

Supported account types

Who can use this application or access this API?

☐ Accounts in this organizational directory only (EID srl only - Single tenant)

☒ Accounts in any organizational directory (Any Azure AD directory - Multitenant)

☐ Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)

☐ Personal Microsoft accounts only

[Help me choose...](#)

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web ▼

https://businesscentral.dynamics.com/OAuthLanding.htm ✓

Register an app you're working on here. Integrate gallery apps and other apps from outside your organization by adding from [Enterprise applications](#).

By proceeding, you agree to the [Microsoft Platform Policies](#) ↗

Register

Figure 13.3: Configuring the app name and supported account type

From the **Overview** pane that appears after the app registration, copy the value of the **Application (client) ID** field (it will be used at a later time):

Home > | App registrations >

PACKTD365BCAPI ✕ ...

Search

«

Delete

Endpoints

Preview features

Overview

Quickstart

Integration assistant

Manage

Branding & properties

Authentication

^ Essentials

Display name : [PACKTD365BCAPI](#)

Application (client) ID : **84a0435b-3ee3-450e-9768-cdaab02cb9a3**

Object ID : 181f9e05-4dd3-4ae9-9840-0902c7736ec9

Directory (tenant) ID :

Supported account types : [Multiple organizations](#)

Client credentials : [Add a certificate or secret](#)

Redirect URIs : [1 web\\_0 spa\\_0 public client](#)

Application ID URI : [Add an Application ID URI](#)

Managed application in L... : [PACKTD365BCAPI](#)

Figure 13.4: Finding the application ID

The app registration is now complete. In the next section, we'll see how to set up the permissions.

## Setting the application permissions

To set the permissions for the newly created application registration, select **API permissions** and then click on **Add a permission**:

Home > App registrations > PACKTD365BCAPI

PACKTD365BCAPI | API permissions

Search Refresh Got feedback?

Overview

Quickstart

Integration assistant

Manage

Branding & properties

Authentication

Certificates & secrets

Token configuration

API permissions

Expose an API

### Configured permissions

Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. [Learn more about permissions and consent](#)

**+ Add a permission** ✓ Grant admin consent for [redacted]

API / Permissions name	Type	Description	Admin consent requ...	Status
▼ Microsoft Graph (1)				
User.Read	Delegated	Sign in and read user profile	No	...


To view and manage consented permissions for individual apps, as well as your tenant's consent settings, try [Enterprise applications](#).

Figure 13.5: Configuring API permissions

From the **Microsoft APIs** list, select **Dynamics 365 Business Central** and then select **Application permissions**. Then assign the **API.ReadWrite.All** permission and select **Add permissions**:

## Request API permissions

[< All APIs](#)



Dynamics 365 Business Central

<https://dynamics.microsoft.com/business-central/overview/> [Docs](#)

What type of permissions does your application require?

Delegated permissions

Your application needs to access the API as the signed-in user.

Application permissions

Your application runs as a background service or daemon without a signed-in user.

Select permissions

expand all

Start typing a permission to filter these results

Permission	Admin consent required
Other permissions	
<input type="checkbox"/> <b>app_access</b> ⓘ Access according to the application's permissions in Dynamics 365 Business Central	Yes
AdminCenter	
<input type="checkbox"/> <b>AdminCenter.ReadWrite.All</b> ⓘ Full access to Admin Center API	Yes
API (1)	
<input checked="" type="checkbox"/> <b>API.ReadWrite.All</b> ⓘ Full access to web services API	Yes

Add permissions

Discard

Figure 13.6: Adding Business Central API permissions

A new permission for Dynamics 365 Business Central will be added. The **Status** field will say **Not granted for the current organization**. The grant will be assigned in Dynamics 365 Business Central:

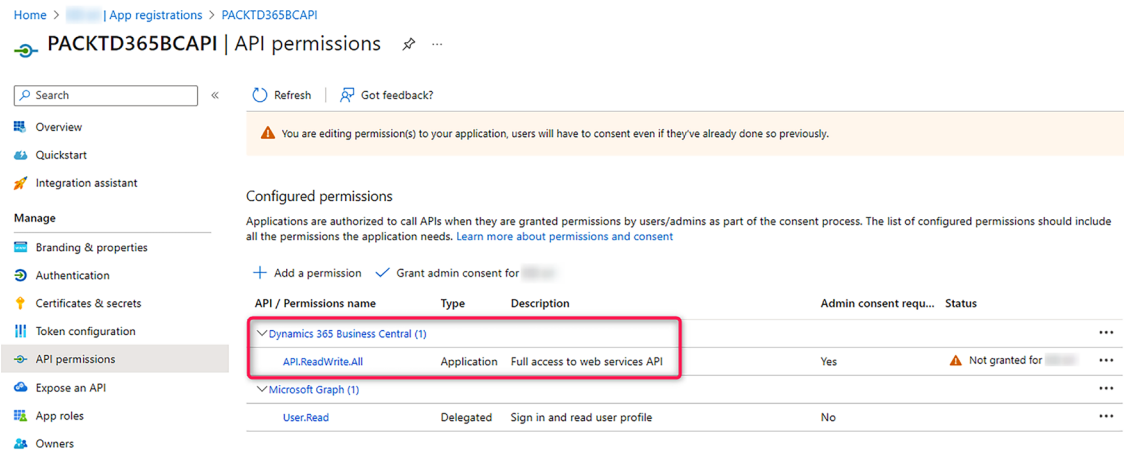


Figure 13.7: Viewing the Business Central API permission

Now we have configured the permissions for our app registration. In the next section, we'll see how to create the secret key.

## Creating a client secret

To create a client secret for your registered application, select **Certificates & secrets** and click on **New client secret**. Then set the secret name and the corresponding due date (maximum 24 months) and click on **Add**:

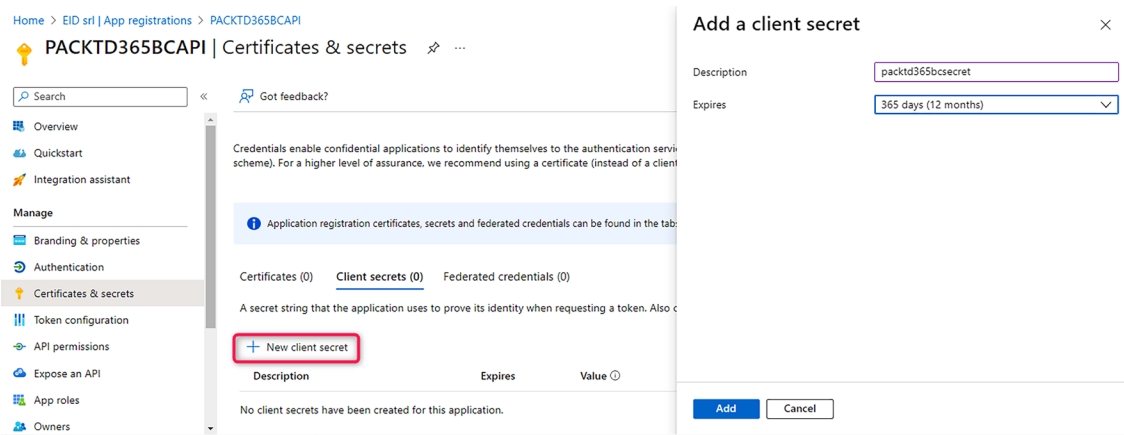



Figure 13.8: Button for adding a client secret

A new secret is generated. Immediately copy the generated secret value (the **Value** field). Its value will never be shown again.



The secret must be renewed when the due date expires. It cannot be set as unlimited expiration.

Certificates (0)   **Client secrets (1)**   Federated credentials (0)

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

+ New client secret

Description	Expires	Value ⓘ	Secret ID
packtd365bcsecret	8/2/2024	Zmz8Q~eenxvBTMjka9DniLI	55c6649f-84fb-449f-8ab0-eb86c

Figure 13.9: Viewing a client secret value

Now we have everything configured. In the next section, we’ll see how to register the application in Dynamics 365 Business Central.

## Microsoft Entra application registration in Dynamics 365 Business Central

Now you’re ready to register your Microsoft Entra application in Dynamics 365 Business Central. To do that, open Dynamics 365 Business Central and search for **Microsoft Entra Applications**. On the **Microsoft Entra Application Card** page, click on **New** and **Create a new record** as follows:

- **Client ID:** Copy the value of the application (client) ID generated from the Azure AD app registration.
- **Description:** Create a description
- **State:** Enabled

In the **User Permission Sets** section, set the API permissions as needed for your scenario. Then select **Grant Consent**.

←

Microsoft Entra Application Card

+

✓ Saved

PACKT D365BC API

Grant Consent

More options

General

Client ID

{84a0435b-3ee3-450e-9768-cd...

Description

PACKT D365BC API

State

Enabled

Contact Information

Extension

App ID

{00000000-0000-0000-0000-00...

App Name

User information

User ID

{bd44f224-157f-4df1-9fa7-b646...

User Name

PACKT D365BC API

You must set the State field to Disabled before you can mak...

User Groups

Manage

Code ↑	Name	Company Name ↑
→		CRONUS IT

Figure 13.10: Granting API permissions

A new login request will appear in order to authorize the app registration. The user that needs to perform the authorization must be a Global Administrator, Application Administrator, or Cloud Application Administrator:

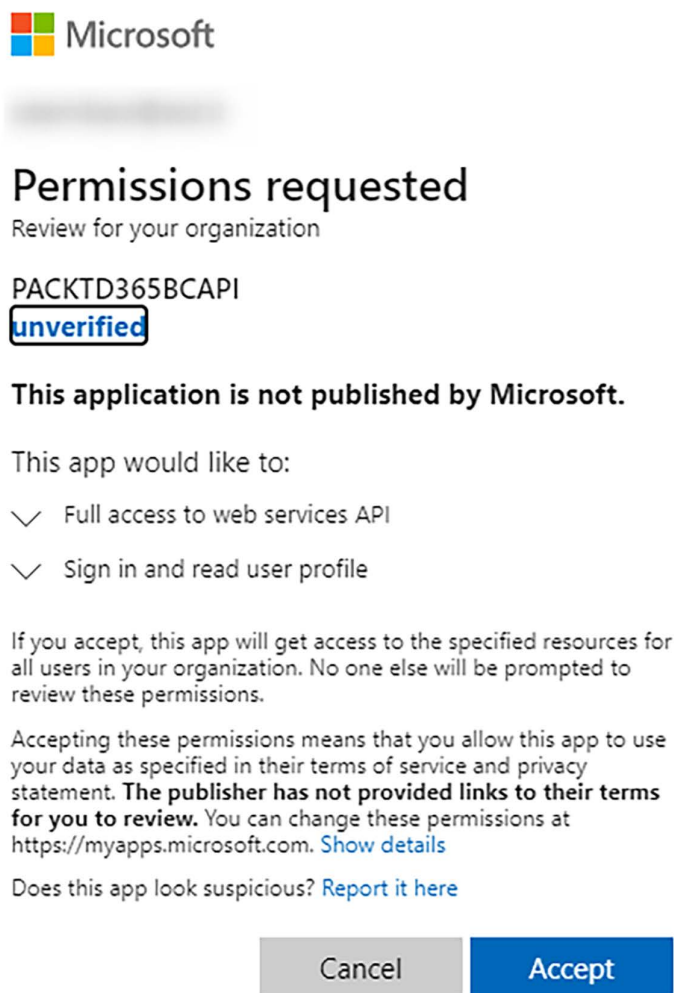


Figure 13.11: Authorizing app registration

Click on **Accept**; if the registration is successfully executed, you will have the following confirmation message:

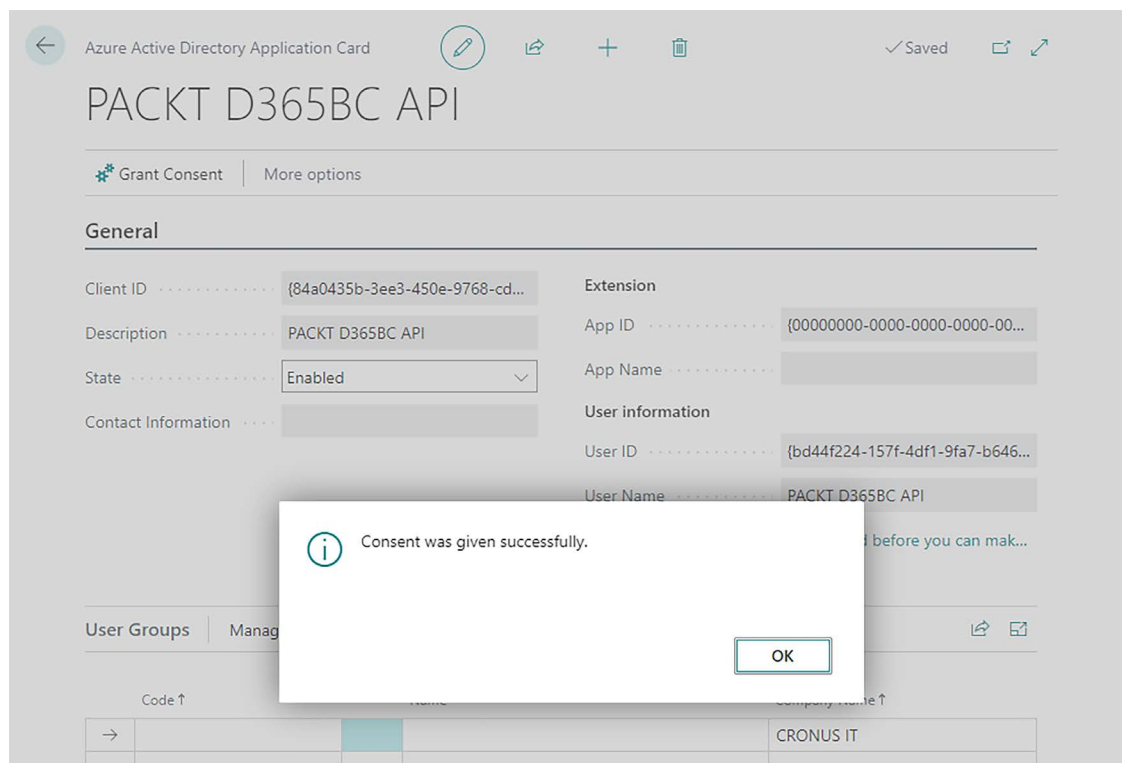


Figure 13.12: Granting app registration

Your Microsoft Entra application is now registered in Dynamics 365 Business Central and you can start using it for making API calls. We'll see that in the next sections.

## Acquiring an authentication token from Microsoft Entra ID

In order to use Dynamics 365 Business Central APIs, you need to request an authentication token to Microsoft Entra ID. An authentication result from Microsoft Entra contains an access token and an ID token. The **access token** is the token we need to use to call Dynamics 365 Business Central APIs (as a Bearer token in the Authorization header of the incoming API call):

```
Authorization: Bearer <token>
```

Remember that an access token is short-lived (one hour by default and one day maximum). When it expires, the client application needs to request a new access token.

Here is an example of HTTP requests that you need to send in order to obtain an authentication token with S2S authentication flow:



I'm using the Rest Client extension from Visual Studio Code to send the needed HTTP requests for the authorization flow. You can obtain the extension from the Visual Studio Code Marketplace or from here: <https://marketplace.visualstudio.com/items?itemName=humao.rest-client>.

```
@tenantId = <tenant id>
@clientId = <client id>
@clientSecret = <client secret>
@baseUri = https://api.businesscentral.dynamics.com
@scope = {{baseUri}}/.default
@bcEnvironmentName = production
@url = {{baseUri}}/v2.0/{{bcEnvironmentName}}/api/v2.0

### Define the Business Central entity name to work with
@EntityName =customers

### Authentication
POST https://login.microsoftonline.com/{{tenantId}}/oauth2/v2.0/token HTTP/1.1
Content-type: application/x-www-form-urlencoded

grant_type=client_credentials
&client_id={{clientId}}
&client_secret={{clientSecret}}
&scope={{scope}}

### Variable Response
@accessHeader = Bearer {{auth.response.body$.access_token}}

### Retrieving Business Central companies
GET {{url}}/companies HTTP/1.1
Authorization: {{accessHeader}}

### Retrieving the company id
@companyId = {{GetCompanies.response.body.value.[0].id}}
@companyUrl = {{url}}/companies/{{companyId}}

### Get entities
GET {{companyUrl}}/{{entityName}} HTTP/1.1
Authorization: {{accessHeader}}
```

For scenarios where you need to use OAuth authentication from AL code (for example, a custom extension that needs to call an external API secured by Microsoft Entra ID authentication), you can use the **OAuth 2.0 module** of the **System** application in the following way:

```
var
    OAuth2: Codeunit OAuth2;
    ClientId: Text;
    ClientSecret: Text;
    MicrosoftOAuth2Url: Text;
    RedirectURL: Text;
    ResourceURL: Label 'https://graph.microsoft.com/';
    AccessToken: Text;

local procedure ClientCredentials()
    var
        Scopes: List of [Text];
    begin
        Scopes.Add(ResourceURL + '.default');

        OAuth2.AcquireTokenWithClientCredentials(ClientId, ClientSecret,
            MicrosoftOAuth2Url, RedirectURL, Scopes, AccessToken);

    end;
```

In this section, you learned how to acquire a security token from Microsoft Entra ID via HTTP calls or via AL Language in your custom extensions. The acquired security token must then be used to authenticate your API calls. In the next section, we'll see how we can use standard Dynamics 365 Business Central APIs.

## Using Dynamics 365 Business Central standard APIs

The Dynamics 365 Business Central platform exposes some standard entities (like Customer and Vendor) as RESTful APIs. The exposed standard entities are listed and documented at the following URL: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/api-reference/v2.0/>.

Dynamics 365 Business Central API endpoints have the following format:

Endpoint URL section	Description
https://api.businesscentral.dynamics.com	Dynamics 365 Business Central base URL (the same for standard and custom APIs)
/v2.0	API version
/your tenant domain	Domain name or ID of the Dynamics 365 Business Central tenant
/environment name	Name of the environment (production, sandbox, and so on). This can be retrieved from the Dynamics 365 Business Central Admin portal
/api	Fixed value
/v2.0	Indicates the version of the API in use

Table 13.1: Endpoint formats

At the time of writing this book, Dynamics 365 Business Central APIs are on endpoint version 2.0, and `API version_number = 2.0`.

You can see the metadata of the exposed OData entities at the following URL:

```
GET https://api.businesscentral.dynamics.com/v2.0/{tenant Id}/{environment name}/api/v2.0/$metadata
```

This call returns an EDMX document that contains a complete description of the feeds, types, properties, and relationships exposed by the service in the **entity data model (EDM)**.

Note that if you’re using OAuth authentication, the tenant ID can also be omitted:

```
GET https://api.businesscentral.dynamics.com/v2.0/{environment name}/api/v1.0/$metadata
```

To start calling Dynamics 365 Business Central APIs, you need to first acquire a security token via OAuth, as explained in the previous section.

When you have the security token, the next thing that you need to work with a Dynamics 365 Business Central entity exposed as an API is a specific company ID (data in Dynamics 365 Business Central is per company). To retrieve the list of available companies on your Dynamics 365 Business Central tenant, you need to send an HTTP GET request to the `/companies` API endpoint. An example of this API call is as follows:

```
GET https://api.businesscentral.dynamics.com/v2.0/<tenantID>/production/api/v2.0/companies
Authorization: Bearer {{token}}
```

This is the response we receive:

```

1  HTTP/1.1 200 OK
2  Transfer-Encoding: chunked
3  Content-Type: application/json; odata.metadata=minimal; odata.streaming=true
4  Content-Encoding: gzip
5  Vary: Accept-Encoding
6  Server: Microsoft-HTTPAPI/2.0
7  Access-Control-Allow-Headers: Origin, X-Requested-With, Authorization
8  Access-Control-Allow-Origin: *
9  Access-Control-Allow-Credentials: true
10 ms-correlation-x: 06e7555b-1ecd-cf84-3db1-f7e287ec4087
11 mise-correlation-id: 688fe97b-5e2f-4e09-9011-9c55ae2a0ff8
12 OData-Version: 4.0
13 Access-Control-Expose-Headers: Date, Content-Length, Server, OData-Version, ms-correlation-x
14 request-id: 81318b12-97e9-4397-9e67-375a3b830a57
15 x-content-type-options: nosniff
16 Strict-Transport-Security: max-age=31536000; includeSubDomains
17 Date: Mon, 17 Jul 2023 15:45:50 GMT
18 Connection: close
19
20 {
21   "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/US Sandbox/api/v2.0/$metadata#companies",
22   "value": [
23     {
24       "id": "18bb64bd-e06c-ed11-81b5-6045bd8e591f",
25       "systemVersion": "22.2.56969.57282",
26       "timestamp": 3807,
27       "name": "CRONUS USA, Inc.",
28       "displayName": "",
29       "businessProfileId": "",
30       "systemCreatedAt": "2022-11-25T16:46:45.96Z",
31       "systemCreatedBy": "00000000-0000-0000-0000-000000000001",
32       "systemModifiedAt": "2022-11-25T16:46:45.96Z",
33       "systemModifiedBy": "00000000-0000-0000-0000-000000000001"
34     },
35     {
36       "id": "989d7ecd-e06c-ed11-81b5-6045bd8e591f",
37       "systemVersion": "22.2.56969.57282",
38       "timestamp": 3808,
39       "name": "My Company",
40       "displayName": "",
41       "businessProfileId": "",
42       "systemCreatedAt": "2022-11-25T16:47:11.433Z",
43       "systemCreatedBy": "00000000-0000-0000-0000-000000000001",
44       "systemModifiedAt": "2022-11-25T16:47:11.433Z",
45       "systemModifiedBy": "00000000-0000-0000-0000-000000000001"
46     }
47   ]
48 }

```

Figure 13.13: Retrieving company IDs with a GET request

If we want to retrieve the list of Customer records for a specific company (for example, Cronus USA, Inc.), we need to send an HTTP GET request to the following API endpoint:

```
GET https://api.businesscentral.dynamics.com/v2.0/<tenantID>/production/api/v2.0/companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/customers
```

```
Authorization: Bearer {{token}}
```

This is the response we receive from it:

```

4 Content-Encoding: gzip
5 Vary: Accept-Encoding
6 Server: Microsoft-HTTPAPI/2.0
7 Access-Control-Allow-Headers: Origin, X-Requested-With, Authorization
8 Access-Control-Allow-Origin: *
9 Access-Control-Allow-Credentials: true
10 ms-correlation-x: 68264f46-6421-16ed-2dcf-62fc29eba995
11 mise-correlation-id: e3e5c2ee-0823-4411-a657-d3719d4eaa61
12 OData-Version: 4.0
13 Access-Control-Expose-Headers: Date, Content-Length, Server, OData-Version, ms-correlation-x
14 request-id: eb5afa59-7689-4cc1-8639-7dbf57ee64f5
15 x-content-type-options: nosniff
16 Strict-Transport-Security: max-age=31536000; includeSubDomains
17 Date: Mon, 17 Jul 2023 15:49:25 GMT
18 Connection: close
19
20 {
21   "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/USSandbox/api/v2.0/$metadata#companies(18b
b64bd-e06c-ed11-81b5-6045bd8e591f)/customers",
22   "value": [
23     {
24       "@odata.etag": "W/\"JzE5OzIwNjM2NDIxODIxNzAwNTI5NjQxOzAwOyc=\"",
25       "id": "71b117ef-e06c-ed11-81b5-6045bd8e591f",
26       "number": "10000",
27       "displayName": "Adatum Corporation",
28       "type": "Company",
29       "addressLine1": "192 Market Square",
30       "addressLine2": "",
31       "city": "Atlanta",
32       "state": "GA",
33       "country": "US",
34       "postalCode": "31772",
35       "phoneNumber": "",
36       "email": "robert.townes@contoso.com",
37       "website": "",
38       "salespersonCode": "JO",
39       "balanceDue": 0,
40       "creditLimit": 0,
41       "taxliable": true,
42       "taxAreaId": "ae7f10f5-e06c-ed11-81b5-6045bd8e591f",
43       "taxAreaDisplayName": "ATLANTA, GA",
44       "taxRegistrationNumber": "",
45       "currencyId": "00000000-0000-0000-0000-000000000000",
46       "currencyCode": "USD",
47       "paymentTermsId": "80b017ef-e06c-ed11-81b5-6045bd8e591f",
48       "shipmentMethodId": "00000000-0000-0000-0000-000000000000",
49       "paymentMethodId": "00000000-0000-0000-0000-000000000000",
50       "blocked": "_x0020_",
51       "lastModifiedDate": "2022-11-25T16:49:20.203Z"
52     },

```

Figure 13.14: Retrieving customer records with a GET request

As you can see, the output (value field) is a JSON array of Customer records.

You can also apply filters when calling the APIs. For example, here, we retrieve all Item records where unitPrice is greater than 100:

```
GET https://api.businesscentral.dynamics.com/v2.0/<tenantID>/production/api/
v2.0/companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/items?$filter=unitPrice%20
gt%20100
Authorization: Bearer {{token}}
```

This is the response:

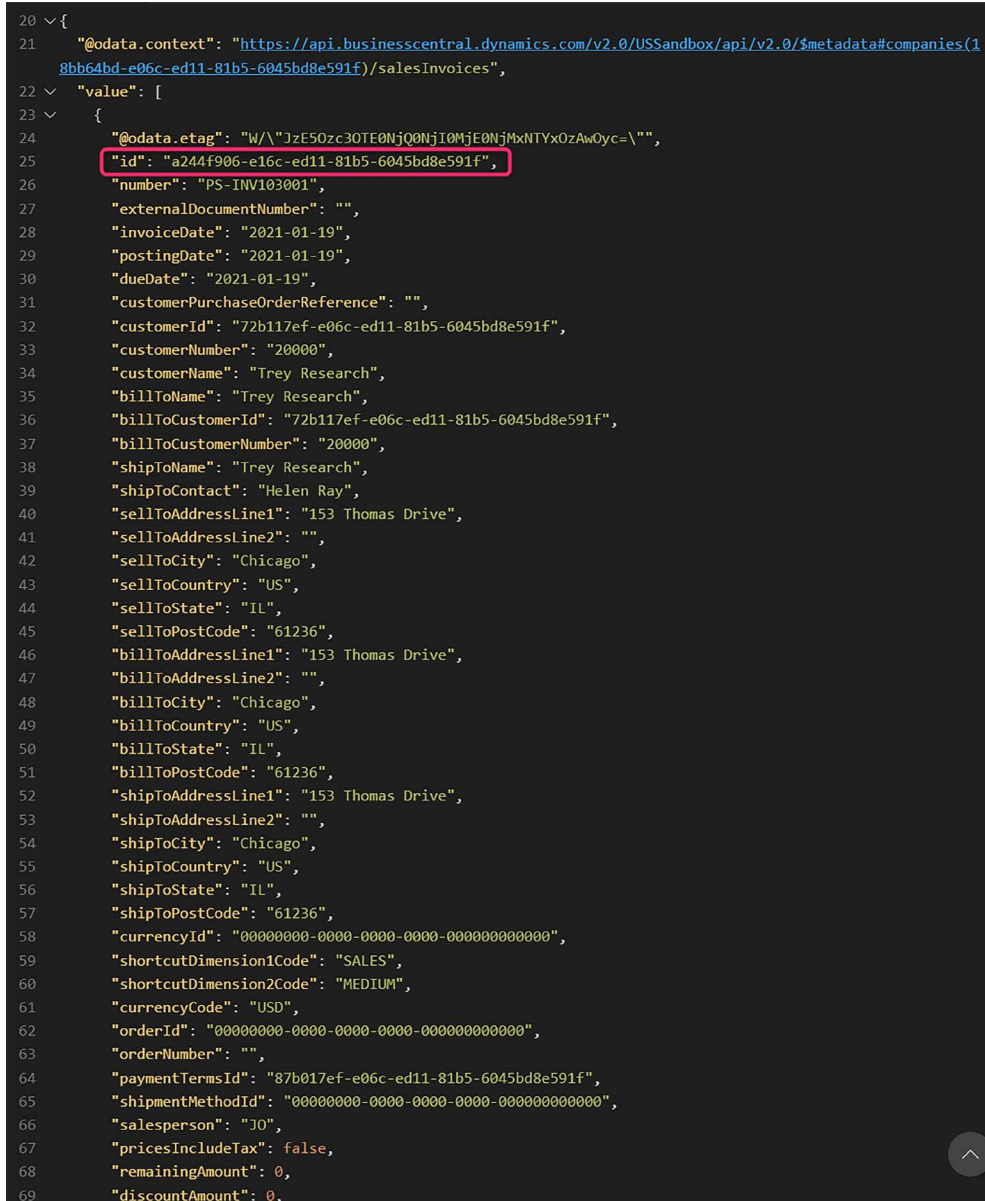
```
1 HTTP/1.1 200 OK
2 Transfer-Encoding: chunked
3 Content-Type: application/json; odata.metadata=minimal; odata.streaming=true
4 Content-Encoding: gzip
5 Vary: Accept-Encoding
6 Server: Microsoft-HTTPAPI/2.0
7 Access-Control-Allow-Headers: Origin, X-Requested-With, Authorization
8 Access-Control-Allow-Origin: *
9 Access-Control-Allow-Credentials: true
10 ms-correlation-x: 3812c1a4-9786-e579-0ba2-91a3a271a29c
11 mise-correlation-id: 934454c4-549b-4bd2-ac82-83b003d62d34
12 OData-Version: 4.0
13 Access-Control-Expose-Headers: Date, Content-Length, Server, OData-Version, ms-correlation-x
14 request-id: 529ba83b-71bd-4f56-8f81-7102ac45c555
15 x-content-type-options: nosniff
16 Strict-Transport-Security: max-age=31536000; includeSubDomains
17 Date: Mon, 17 Jul 2023 15:53:38 GMT
18 Connection: close
19
20 {
21   "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/US Sandbox/api/v2.0/$metadata#companies(18b
22   b64bd-e06c-ed11-81b5-6045bd8e591f)/items",
23   "value": [
24     {
25       "@odata.etag": "W/\"JzIwOzE2OTA3ODI1MDQzMTYyMzYzODEzMTswMDsn\"",
26       "id": "7bb117ef-e06c-ed11-81b5-6045bd8e591f",
27       "number": "1896-S",
28       "displayName": "ATHENS Desk",
29       "type": "Inventory",
30       "itemCategoryId": "478110f5-e06c-ed11-81b5-6045bd8e591f",
31       "itemCategoryCode": "TABLE",
32       "blocked": false,
33       "gtin": "",
34       "inventory": 4,
35       "unitPrice": 1000.8,
36       "priceIncludesTax": false,
37       "unitCost": 780.7,
38       "taxGroupId": "be7f10f5-e06c-ed11-81b5-6045bd8e591f",
39       "taxGroupCode": "FURNITURE",
40       "baseUnitOfMeasureId": "c77d10f5-e06c-ed11-81b5-6045bd8e591f",
41       "baseUnitOfMeasureCode": "PCS",
42       "generalProductPostingGroupId": "3b7f10f5-e06c-ed11-81b5-6045bd8e591f",
43       "generalProductPostingGroupCode": "RETAIL",
44       "inventoryPostingGroupId": "b67d10f5-e06c-ed11-81b5-6045bd8e591f",
45       "inventoryPostingGroupCode": "RESALE",
46       "lastModifiedDateTime": "2022-11-25T16:52:34.62Z"
47     }
48   ]
49 }
```

Figure 13.15: Retrieving item records with a GET request

To have the list of sales invoices, you can perform a GET request to the following endpoint:

```
GET https://api.businesscentral.dynamics.com/v2.0/<tenantID>/production/api/
v2.0/companies(18bb64bd-e06c-ed11-81b5-604
5bd8e591f)/salesInvoices
Authorization: Bearer {{token}}
```

The following screenshots shows the results of this request:



```
20 {
21   "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/US$andbox/api/v2.0/$metadata#companies(1
8bb64bd-e06c-ed11-81b5-6045bd8e591f)/salesInvoices",
22   "value": [
23     {
24       "@odata.etag": "W/\"JzE50zc30TE0NjQ0NjI0MjE0NjMxNTYxOzAwOyc=\"",
25       "id": "a244f906-e16c-ed11-81b5-6045bd8e591f",
26       "number": "PS-INV103001",
27       "externalDocumentNumber": "",
28       "invoiceDate": "2021-01-19",
29       "postingDate": "2021-01-19",
30       "dueDate": "2021-01-19",
31       "customerPurchaseOrderReference": "",
32       "customerId": "72b117ef-e06c-ed11-81b5-6045bd8e591f",
33       "customerNumber": "20000",
34       "customerName": "Trey Research",
35       "billToName": "Trey Research",
36       "billToCustomerId": "72b117ef-e06c-ed11-81b5-6045bd8e591f",
37       "billToCustomerNumber": "20000",
38       "shipToName": "Trey Research",
39       "shipToContact": "Helen Ray",
40       "sellToAddressLine1": "153 Thomas Drive",
41       "sellToAddressLine2": "",
42       "sellToCity": "Chicago",
43       "sellToCountry": "US",
44       "sellToState": "IL",
45       "sellToPostCode": "61236",
46       "billToAddressLine1": "153 Thomas Drive",
47       "billToAddressLine2": "",
48       "billToCity": "Chicago",
49       "billToCountry": "US",
50       "billToState": "IL",
51       "billToPostCode": "61236",
52       "shipToAddressLine1": "153 Thomas Drive",
53       "shipToAddressLine2": "",
54       "shipToCity": "Chicago",
55       "shipToCountry": "US",
56       "shipToState": "IL",
57       "shipToPostCode": "61236",
58       "currencyId": "00000000-0000-0000-0000-000000000000",
59       "shortcutDimension1Code": "SALES",
60       "shortcutDimension2Code": "MEDIUM",
61       "currencyCode": "USD",
62       "orderId": "00000000-0000-0000-0000-000000000000",
63       "orderNumber": "",
64       "paymentTermsId": "87b017ef-e06c-ed11-81b5-6045bd8e591f",
65       "shipmentMethodId": "00000000-0000-0000-0000-000000000000",
66       "salesperson": "JO",
67       "pricesIncludeTax": false,
68       "remainingAmount": 0,
69       "discountAmount": 0,
```

Figure 13.16: Retrieving sales invoices with a GET request

Dynamics 365 Business Central standard APIs also support features such as **expand**, in which, in a single call, you can expand the relationships between entities, and retrieve the main entity along with the related entities. For example, to retrieve a sales invoice and all of its sales invoice line records in a single HTTP call, you can perform an HTTP GET call to the following API endpoint:

```
GET https://api.businesscentral.dynamics.com/v2.0/<tenantID>/production/api/v2.0/companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/salesInvoices(a244f906-e16c-ed11-81b5-6045bd8e591f)?$expand=salesInvoiceLines
Authorization: Bearer {{token}}
```

As a result, you have a single JSON response object with the sales invoice header and its related sales invoice line detail. Here is the header object:

```
20 {
21   "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/US Sandbox/api/v2.0/$metadata#companies(18b
22     b64bd-e06c-ed11-81b5-6045bd8e591f)/salesInvoices/$entity",
23   "@odata.etag": "W/\"JzE5Ozc3OTE0NjI0MjE0NjE0NjE0NjE0NTYxOzAwOyc=\"",
24   "id": "a244f906-e16c-ed11-81b5-6045bd8e591f",
25   "number": "PS-INV103001",
26   "externalDocumentNumber": "",
27   "invoiceDate": "2021-01-19",
28   "postingDate": "2021-01-19",
29   "dueDate": "2021-01-19",
30   "customerPurchaseOrderReference": "",
31   "customerId": "72b117ef-e06c-ed11-81b5-6045bd8e591f",
32   "customerNumber": "20000",
33   "customerName": "Trey Research",
34   "billToName": "Trey Research",
35   "billToCustomerId": "72b117ef-e06c-ed11-81b5-6045bd8e591f",
36   "billToCustomerNumber": "20000",
37   "shipToName": "Trey Research",
38   "shipToContact": "Helen Ray",
39   "sellToAddressLine1": "153 Thomas Drive",
40   "sellToAddressLine2": "",
41   "sellToCity": "Chicago",
42   "sellToCountry": "US",
43   "sellToState": "IL",
44   "sellToPostCode": "61236",
45   "billToAddressLine1": "153 Thomas Drive",
46   "billToAddressLine2": "",
47   "billToCity": "Chicago",
48   "billToCountry": "US",
49   "billToState": "IL",
50   "billToPostCode": "61236",
51   "shipToAddressLine1": "153 Thomas Drive",
52   "shipToAddressLine2": "",
53   "shipToCity": "Chicago",
54   "shipToCountry": "US",
55   "shipToState": "IL",
56   "shipToPostCode": "61236",
57   "currencyId": "00000000-0000-0000-0000-000000000000",
58   "shortcutDimension1Code": "SALES",
59   "shortcutDimension2Code": "MEDIUM",
60   "currencyCode": "USD",
61   "orderId": "00000000-0000-0000-0000-000000000000",
62   "orderNumber": "",
63   "paymentTermsId": "87b017ef-e06c-ed11-81b5-6045bd8e591f",
64   "shipmentMethodId": "00000000-0000-0000-0000-000000000000",
65   "salesperson": "JO",
66   "pricesIncludeTax": false,
67   "remainingAmount": 0,
68   "discountAmount": 0,
69   "discountAppliedBeforeTax": true,
```

Figure 13.17: Retrieving a sales invoice header with a GET request

Also, here are the related line's details:

```

59  "currencyCode": "USD",
60  "orderId": "00000000-0000-0000-0000-000000000000",
61  "orderNumber": "",
62  "paymentTermsId": "87b017ef-e06c-ed11-81b5-6045bd8e591f",
63  "shipmentMethodId": "00000000-0000-0000-0000-000000000000",
64  "salesperson": "JO",
65  "pricesIncludeTax": false,
66  "remainingAmount": 0,
67  "discountAmount": 0,
68  "discountAppliedBeforeTax": true,
69  "totalAmountExcludingTax": 164.7,
70  "totalTaxAmount": 8.24,
71  "totalAmountIncludingTax": 172.94,
72  "status": "Paid",
73  "lastModifiedDateTime": "2022-11-25T16:50:27.647Z",
74  "phoneNumber": "",
75  "email": "helen.ray@contoso.com",
76  "salesInvoiceLines": [
77    {
78      "@odata.etag": "W/\\"JzIwOzE1OTY1Nzc0NTM2MDI5Mzg0NzQ3MTswMDsn\"",
79      "id": "e0a67029-e16c-ed11-81b5-6045bd8e591f",
80      "documentId": "a244f906-e16c-ed11-81b5-6045bd8e591f",
81      "sequence": 10000,
82      "itemId": "81b117ef-e06c-ed11-81b5-6045bd8e591f",
83      "accountId": "00000000-0000-0000-0000-000000000000",
84      "lineType": "Item",
85      "lineObjectNumber": "1928-S",
86      "description": "AMSTERDAM Lamp",
87      "unitOfMeasureId": "c77d10f5-e06c-ed11-81b5-6045bd8e591f",
88      "unitOfMeasureCode": "PCS",
89      "quantity": 3,
90      "unitPrice": 54.9,
91      "discountAmount": 0,
92      "discountPercent": 0,
93      "discountAppliedBeforeTax": false,
94      "amountExcludingTax": 164.7,
95      "taxCode": "FURNITURE",
96      "taxPercent": 5,
97      "totalTaxAmount": 8.24,
98      "amountIncludingTax": 172.94,
99      "invoiceDiscountAllocation": 0,
100     "netAmount": 164.7,
101     "netTaxAmount": 8.24,
102     "netAmountIncludingTax": 172.94,
103     "shipmentDate": "2022-11-25",
104     "itemVariantId": "00000000-0000-0000-0000-000000000000",
105     "locationId": "00000000-0000-0000-0000-000000000000"
106   }
107 ]
108 }

```

Figure 13.18: Retrieving sales invoice line details with a GET request

Dynamics 365 Business Central APIs also support the \$select OData clause, which permits you to retrieve only the specified fields of a record set. For example, we can modify the previous API call to retrieve the sales invoice details by specifying only a small set of fields that we need in the following way:

```
GET https://api.businesscentral.dynamics.com/v2.0/<tenantID>/production/
api/v2.0/companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/salesInvoices
?$select=id,number,invoiceDate,customerId,customerNumber,customerName
Authorization: Bearer {{token}}
```

And the response will be the following JSON:

```
{
  "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/US Sandbox/api/v2.0/$metadata#companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/salesInvoices",
  "value": [
    {
      "@odata.etag": "W/\"JzE50zc30TE0NjQ0NjI0MjE0NjMxNTYxOzAwOyc=\"",
      "id": "a244f906-e16c-ed11-81b5-6045bd8e591f",
      "number": "PS-INV103001",
      "invoiceDate": "2021-01-19",
      "customerId": "72b117ef-e06c-ed11-81b5-6045bd8e591f",
      "customerNumber": "20000",
      "customerName": "Trey Research"
    },
    {
      "@odata.etag": "W/\"JzIwOzE3MTY5MzQ4ODU1MzA1MDczMDA0MTswMDsn\"",
      "id": "a444f906-e16c-ed11-81b5-6045bd8e591f",
      "number": "PS-INV103002",
      "invoiceDate": "2021-01-20",
      "customerId": "71b117ef-e06c-ed11-81b5-6045bd8e591f",
      "customerNumber": "10000",
      "customerName": "Adatum Corporation"
    },
    {
      "@odata.etag": "W/\"JzE50zg1MzIyMDE2ODMzMzk3ODIwMTgxOzAwOyc=\"",
      "id": "a644f906-e16c-ed11-81b5-6045bd8e591f",
      "number": "PS-INV103003",
      "invoiceDate": "2021-01-21",
      "customerId": "73b117ef-e06c-ed11-81b5-6045bd8e591f",
      "customerNumber": "30000",
      "customerName": "School of Fine Art"
    },
    {
      "@odata.etag": "W/\"JzE50zg1MTYxNzk2NDI4MDYyMzQ0MjQxOzAwOyc=\"",
      "id": "a844f906-e16c-ed11-81b5-6045bd8e591f",
      "number": "PS-INV103004",
      "invoiceDate": "2021-01-22",
      "customerId": "75b117ef-e06c-ed11-81b5-6045bd8e591f",
      "customerNumber": "50000",
      "customerName": "Relecloud"
    }
  ]
}
```

Figure 13.19: Retrieving specific lines with GET request

After calling a Dynamics 365 Business Central API, you can then parse the JSON response as per your needs.

In the next section, we will see how to create a custom API page for a new entity added to Dynamics 365 Business Central and how to create a new API page for an existing entity.

# Creating a custom API in Dynamics 365 Business Central

With Dynamics 365 Business Central extensions, you can create custom entities and you can also expose those custom entities as RESTful APIs. Custom entities are entities that don't exist in the standard product, which will be useful for you when you need an entity for a particular project or a particular customization.

To create a new API in Dynamics 365 Business Central, you need to define a new Page object with PageType = API. To do this, you can use the tpage snippet and then select **Page of type API**, as follows:

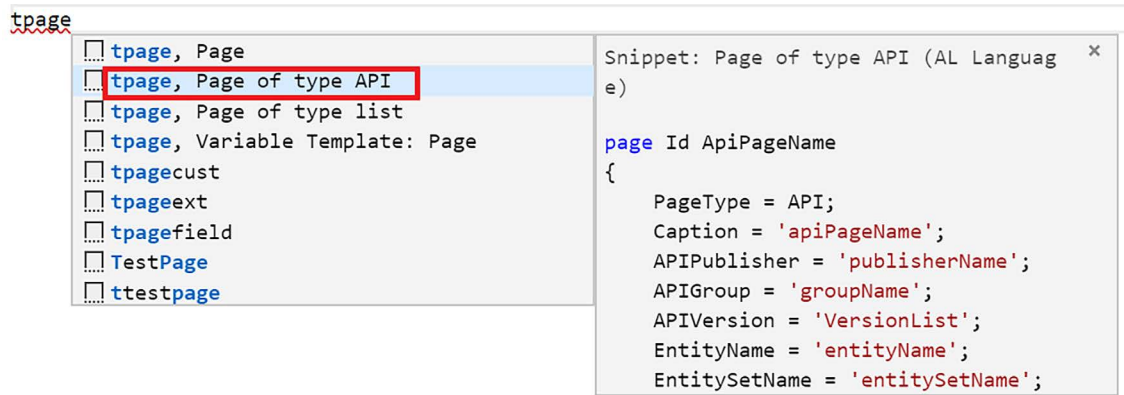


Figure 13.20: Defining a new Page object

When creating API pages, remember the following:

- Fields must have a name in the REST API-compliant format (only alphanumeric values, and no spaces or special characters (camelCase)).
- You should always expose the ID of the entity (SystemId) and this is the recommended OData key to use.
- When you insert, modify, or delete an entity through APIs, triggers on the underlying table are not executed. You need to call the table's trigger by handling the corresponding triggers at the page level.
- If needed, in the OnModify trigger of the API page, you need to handle the possibility of renaming a record (an API call via a record ID can issue a primary key rename).

Here, we'll see two main scenarios:

- How to implement an API for a custom entity (assuming that there's an extension that adds the Car entity to Dynamics 365 Business Central for managing car details inside the ERP)
- How to implement a new API for an existing entity

We will look into each of these scenarios in the following sections.

## Implementing a new API for a custom entity

In this example, we will be creating a new entity in Dynamics 365 Business Central to handle the details of cars, and this entity will also be exposed as an API for external applications:

1. To do this, we first create a new Car table, as follows:

```
table 50200 Car
{
    DataClassification = CustomerContent;
    Caption = 'Car';
    LookupPageId = "Car List";
    DrillDownPageId = "Car List";
    fields
    {
        field(1; ModelNo; Code[20])
        {
            Caption = 'Model No.';
            DataClassification = CustomerContent;
        }
        field(2; Description; Text[100])
        {
            Caption = 'Description';
            DataClassification = CustomerContent;
        }
        field(3; Brand; Code[20])
        {
            Caption = 'Brand';
            DataClassification = CustomerContent;
        }
        field(4; Power; Integer)
        {
            Caption = 'Power (CV)';
            DataClassification = CustomerContent;
        }
        field(5; "Engine Type"; Enum EngineType)
        {
            Caption = 'Engine Type';
            DataClassification = CustomerContent;
        }
    }
}
```

```

    keys
    {
        key(PK; ModelNo)
        {
            Clustered = true;
        }
    }
}

```

2. The Engine Type field is of the Enum EngineType type, and the enum is defined as follows:

```

enum 50200 EngineType
{
    Extensible = true;
    value(0; Petrol)
    {
        Caption = 'Petrol';
    }
    value(1; Diesel)
    {
        Caption = 'Diesel';
    }
    value(2; Electric)
    {
        Caption = 'Electric';
    }
    value(3; Hybrid)
    {
        Caption = 'Hybrid';
    }
}

```

3. We also create a Car List page (a standard list page) for managing Car data in Dynamics 365 Business Central.
4. Now, we need to create the API page (by using the tpage snippet and then selecting **Page of type API**). The CarAPI page is defined as follows:

```

page 50200 CarAPI
{
    PageType = API;
    Caption = 'Car API';
    APIPublisher = 'sd';
}

```

```

APIGroup = 'custom';
APIVersion = 'v1.0';
EntityName = 'car';
EntitySetName = 'cars';
SourceTable = Car;
DelayedInsert = true;
ODataKeyFields = systemId;

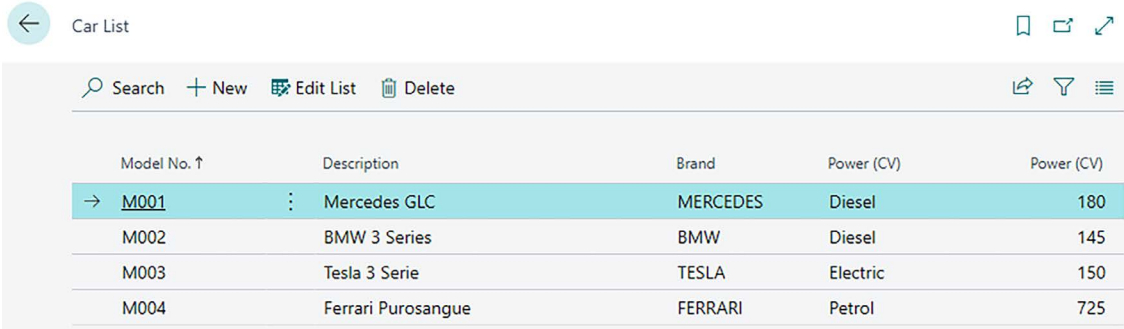
layout
{
    area(Content)
    {
        repeater(GroupName)
        {
            field(modelno; ModelNo)
            {
                Caption = 'modelNo', Locked = true;
            }
            field(description; Description)
            {
                Caption = 'description', Locked = true;
            }
            field(brand; Brand)
            {
                Caption = 'brand', Locked = true;
            }
            field(engineType; "Engine Type")
            {
                Caption = 'engineType', Locked = true;
            }
            field(power; Power)
            {
                Caption = 'power', Locked = true;
            }
            field(systemId; Rec.SystemId)
            {
                Caption = 'systemId', Locked = true;
            }
        }
    }
}

```

This page exposes the fields that we want to have in our API by applying the naming rules according to the OData specifications. Here, you can see that we have defined some important parameters:

- **APIPublisher:** Sets the publisher of the API endpoint that the page is exposed in
- **APIGroup:** Sets the group of the API endpoint that the page is exposed in
- **APIVersion:** Sets the version(s) of the API endpoint the page is exposed in
- **EntityName:** Sets the singular entity name with which the page is exposed in the API endpoint
- **EntitySetName:** Sets the plural entity name with which the page is exposed in the API endpoint
- **SourceTable:** Sets the table name from which this page will display records
- **ODataKeyFields:** Specifies the fields to select when using OData

Now, press F5 in Visual Studio Code and publish your extension. When it's published, search for Car List and then insert some example Car records, such as the following:



The screenshot shows a web application titled 'Car List'. It features a search bar, a '+ New' button, and buttons for 'Edit List' and 'Delete'. Below the navigation bar is a table with five columns: 'Model No. ↑', 'Description', 'Brand', 'Power (CV)', and 'Power (CV)'. The table contains four records. The first record, 'M001 Mercedes GLC', is highlighted in light blue. A right arrow icon is visible to the left of the first record's model number.

Model No. ↑	Description	Brand	Power (CV)	Power (CV)
→ M001	Mercedes GLC	MERCEDES	Diesel	180
M002	BMW 3 Series	BMW	Diesel	145
M003	Tesla 3 Serie	TESLA	Electric	150
M004	Ferrari Purosangue	FERRARI	Petrol	725

Figure 13.21: Inserting Car records

Now, we can test our custom API. When published in a SaaS tenant, a custom API endpoint has the following format:

```
{BaseURL}/api/<api publisher>/<api group>/<api version>
```

Here, {BaseURL} is `https://api.businesscentral.dynamics.com/v2.0/ENVIRONMENTNAME`.

If you're testing it on a Docker-based sandbox or in an on-premises environment, the API endpoint is like this:

```
{ServerUrl:ODATA_Port}/{ServerInstance}/api//<api publisher>/<api group>/<api version>
```

The APIs are invoked per company, as previously explained, and you need to first retrieve the company ID of the desired company.

To have the list of cars for the selected company, you need to send a GET request to the following URL (by passing the company ID that was retrieved with the preceding call):

```
{baseUrl}/api/sd/custom/v1.0/companies({id})/cars
```

This is the response that we get:

```

{
  "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/USSandbox/api/sd/custom/v1.0/$metadata#comp
anies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/cars",
  "value": [
    {
      "@odata.etag": "W/\"JzE50zg10TQ5MTY1ODQ0OTg5NjYxNzQxOzAwOyc=\"",
      "systemId": "24e5343d-a927-ee11-bdf5-00224828e836",
      "modelName": "M001",
      "description": "Mercedes GLC",
      "brand": "MERCEDES",
      "engineType": "Diesel",
      "power": 180
    },
    {
      "@odata.etag": "W/\"JzE50zgzMzQ2ODgzNDQzOTM0MzUzMTYxOzAwOyc=\"",
      "systemId": "f91e104d-a927-ee11-bdf5-00224828e836",
      "modelName": "M002",
      "description": "BMW 3 Serie",
      "brand": "BMW",
      "engineType": "Diesel",
      "power": 145
    },
    {
      "@odata.etag": "W/\"JzE50zMyNjg4Mzc4NjYzNzUyMjQyNTEwOzAwOyc=\"",
      "systemId": "0528e058-a927-ee11-bdf5-00224828e836",
      "modelName": "M003",
      "description": "Tesla 3 Serie",
      "brand": "TESLA",
      "engineType": "Electric",
      "power": 150
    },
    {
      "@odata.etag": "W/\"JzE40zM5NjAyNDQxNTQxNTE7MDA7Jw==\"",
      "systemId": "e1df7062-a927-ee11-bdf5-00224828e836",
      "modelName": "M004",
      "description": "Ferrari Purosangue",
      "brand": "FERRARI",
      "engineType": "Petrol",
      "power": 725
    }
  ]
}

```

Figure 13.22: Retrieving Car records with a GET request

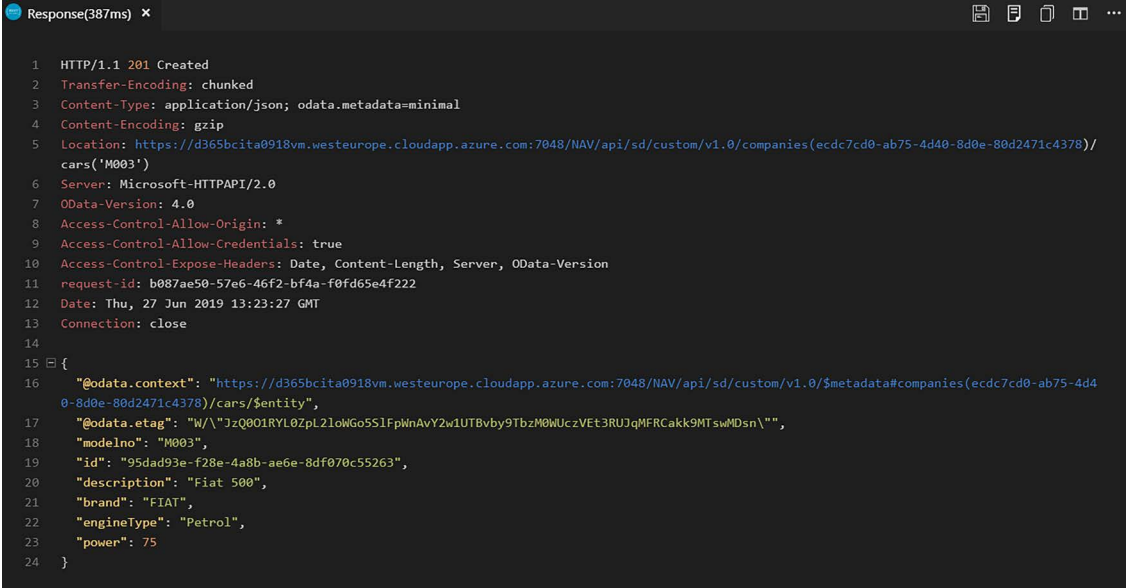
As you can see, we have the JSON representation of the inserted records, and every field (JSON token) has the name that we assigned in our API definition.

To insert a new Car record via our previously published custom cars API, you need to send a POST request to the following URL by passing the JSON of the Car record to create in the body:

```
POST {{baseurl}}/api/sd/custom/v1.0/companies({{companyid}})/cars
Content-Type: application/json
Authorization: Bearer {{token}}

{
  "modelno": "M005",
  "description": "Fiat 500",
  "brand": "FIAT",
  "engineType": "Petrol",
  "power": 75
}
```

The response received is as follows:



```
Response(387ms) X
1 HTTP/1.1 201 Created
2 Transfer-Encoding: chunked
3 Content-Type: application/json; odata.metadata=minimal
4 Content-Encoding: gzip
5 Location: https://d365bcita0918vm.westeurope.cloudapp.azure.com:7048/NAV/api/sd/custom/v1.0/companies(ecdc7cd0-ab75-4d40-8d0e-80d2471c4378)/cars('M003')
6 Server: Microsoft-HTTPAPI/2.0
7 OData-Version: 4.0
8 Access-Control-Allow-Origin: *
9 Access-Control-Allow-Credentials: true
10 Access-Control-Expose-Headers: Date, Content-Length, Server, OData-Version
11 request-id: b087ae50-57e6-46f2-bf4a-f0fd65e4f222
12 Date: Thu, 27 Jun 2019 13:23:27 GMT
13 Connection: close
14
15 {
16   "@odata.context": "https://d365bcita0918vm.westeurope.cloudapp.azure.com:7048/NAV/api/sd/custom/v1.0/$metadata#companies(ecdc7cd0-ab75-4d40-8d0e-80d2471c4378)/cars/$entity",
17   "@odata.etag": "W/\"JzQ001RYL0ZpL2loWGo5S1FpWnAvY2w1UTBvby9TbzM0WUc2VET3RUJqMFRkCakk9MTswMDsn\"",
18   "modelno": "M003",
19   "id": "95dad93e-f28e-4a8b-ae6e-8df070c55263",
20   "description": "Fiat 500",
21   "brand": "FIAT",
22   "engineType": "Petrol",
23   "power": 75
24 }
```

Figure 13.23: Inserting Car records with a POST request

We receive HTTP/1.1 201 Created and the JSON details of the Car record are added to Dynamics 365 Business Central.

If you look at Car List in Dynamics 365 Business Central, you can see that a new record has been created:

MODEL NO.	DESCRIPTION	BRAND	POWER (CV)	POWER (CV)
M001	Mercedes GLC	MERCEDES	Diesel	180
M002	BMW 3 Series	BMW	Diesel	145
M003	Fiat 500	FIAT	Petrol	75

Figure 13.24: Viewing the new record



When sending a POST request, remember to correctly set the content type of the request to `application/json`. Otherwise, you can receive a quite confusing error in the response message, such as `{"error":{"code":"BadRequest","message":"Cannot create an instance of an interface."}}`.

To retrieve the details of a specific car record, just send a GET request to the following URL:

```
{baseUrl}/api/sd/custom/v1.0/companies({id})/cars({id})
```

This is done by passing the GUID of the car record to retrieve.

In our example, if we want to retrieve the details of the Mercedes record, we have to send an HTTP GET request to the following URL:

```
{BaseUrl}/api/sd/custom/v1.0/companies(CompanyID)/cars(24e5343d-a927-ee11-bdf5-00224828e836)
```

Here, 24e5343d-a927-ee11-bdf5-00224828e836 is the `SystemId` of the record we want to retrieve. This is the response we receive:

```

HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true
Content-Encoding: gzip
Vary: Accept-Encoding
Server: Microsoft-HTTPAPI/2.0
Access-Control-Allow-Headers: Origin, X-Requested-With, Authorization
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
ms-correlation-x: 00ad5e39-775e-5c58-dd15-7f4e3019a41d
mise-correlation-id: 4aeb62f1-02fd-47b4-9c0d-3816928891ab
OData-Version: 4.0
Access-Control-Expose-Headers: Date, Content-Length, Server, OData-Version, ms-correlation-x
request-id: 7892e2de-cba2-468f-bd75-5c79003d7883
x-content-type-options: nosniff
Strict-Transport-Security: max-age=31536000; includeSubDomains
Date: Wed, 26 Jul 2023 09:07:11 GMT
Connection: close

{
  "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/US Sandbox/api/sd/custom/v1.0/$metadata#companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/cars/$entity",
  "@odata.etag": "W/\"JzE50zg10TQ5MTY1ODQ0TG5NjYxNzQxOzAwOyc=\"",
  "systemId": "24e5343d-a927-ee11-bdf5-00224828e836",
  "modelName": "M001",
  "description": "Mercedes GLC",
  "brand": "MERCEDES",
  "engineType": "Diesel",
  "power": 180
}

```

Figure 13.25: Retrieving Car record details with a GET request

As you can see, we have retrieved the JSON representation of the specified Car record.

## Implementing a new API for an existing entity

As we discussed in the *Using the OData protocol for APIs* section, you cannot extend an existing standard Dynamics 365 Business Central API page. If you need to retrieve new fields for a standard Dynamics 365 Business Central entity, you need to create a new API page in your namespace.

For example, here, I am creating a simple new API that retrieves a customer's details, which are not natively exposed in the standard Customer API. The API page is defined as follows:

```

page 50115 PKTCustomerAPI
{
    PageType = API;
    Caption = 'customer';
    APIPublisher = 'PACKT';
}

```

```
APIVersion = 'v1.0';
APIGroup = 'customapi';
EntityName = 'customer';
EntitySetName = 'customers';
SourceTable = Customer;
DelayedInsert = true;
ODataKeyFields = SystemId;

layout
{
    area(Content)
    {
        repeater(GroupName)
        {
            field(no; Rec."No.")
            {
                Caption = 'no', Locked = true;
            }
            field(name; Rec.Name)
            {
                Caption = 'name', Locked = true;
            }
            field(Id; Rec.SystemId)
            {
                Caption = 'Id', Locked = true;
            }
            field(balanceDue; Rec."Balance Due")
            {
                Caption = 'balanceDue', Locked = true;
            }
            field(creditLimit; Rec."Credit Limit (LCY)")
            {
                Caption = 'creditLimit', Locked = true;
            }
            field(currencyCode; Rec."Currency Code")
            {
                Caption = 'currencyCode', Locked = true;
            }
            field(email; Rec."E-Mail")
            {
                Caption = 'email', Locked = true;
            }
        }
    }
}
```

```

    }
    field(balance; Rec."Balance (LCY)")
    {
        Caption = 'balance', Locked = true;
    }
    field(countryRegionCode; Rec."Country/Region Code")
    {
        Caption = 'countryRegionCode', Locked = true;
    }
    field(netChange; Rec."Net Change")
    {
        Caption = 'netChange', Locked = true;
    }
    field(noOfOrders; Rec."No. of Orders")
    {
        Caption = 'noOfOrders', Locked = true;
    }
    field(noOfReturnOrders; Rec."No. of Return Orders")
    {
        Caption = 'noOfReturnOrders', Locked = true;
    }
    field(phoneNo; Rec."Phone No.")
    {
        Caption = 'phoneNo', Locked = true;
    }
    field(salesLCY; Rec."Sales (LCY)")
    {
        Caption = 'salesLCY', Locked = true;
    }
    field(shippedNotInvoiced; Rec."Shipped Not Invoiced")
    {
        Caption = 'shippedNotInvoiced', Locked = true;
    }
    }
}
}
}

```

When this is published on our Dynamics 365 Business Central tenant, we can reach this API at the following endpoint:

```
{baseUrl}/api/PACKT/customapi/v1.0/companies({id})/customers
```

If we send an HTTP GET request to this endpoint to retrieve the Customer records, we get the following API response:

```

{
  "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/USSandbox/api/packt/customapi/v1.0/$metadata#companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/customers",
  "value": [
    {
      "@odata.etag": "W/\"JzIwOzEwMDIzOTQ3MDC4OTQzNTgyMjk5MTswMDsn\"",
      "Id": "71b117ef-e06c-ed11-81b5-6045bd8e591f",
      "no": "10000",
      "name": "Adatum Corporation",
      "balanceDue": 0,
      "creditLimit": 0,
      "currencyCode": "",
      "email": "robert.townes@contoso.com",
      "balance": 0,
      "countryRegionCode": "US",
      "netChange": 0,
      "noOfOrders": 2,
      "noOfReturnOrders": 0,
      "phoneNo": "",
      "salesLCY": 223598.4,
      "shippedNotInvoiced": 0
    },
    {
      "@odata.etag": "W/\"JzE5OzE0NjM0MDYwMTIzMDUyMTgwNzcxOzAwOyc=\"\"",
      "Id": "72b117ef-e06c-ed11-81b5-6045bd8e591f",
      "no": "20000",
      "name": "Trey Research",
      "balanceDue": 3036.6,
      "creditLimit": 0,
      "currencyCode": "",
      "email": "helen.ray@contoso.com",
      "balance": 3036.6,
      "countryRegionCode": "US",
      "netChange": 3036.6,
      "noOfOrders": 0,
      "noOfReturnOrders": 0,
      "phoneNo": "",
      "salesLCY": 58673,
      "shippedNotInvoiced": 0
    }
  ]
}

```

Figure 13.26: Retrieving customer records with a GET request using the custom API

As you can see, the custom API shows all of the Customer fields we have added to our API page (the Normal and FlowFields fields).

A standard API page (like the one just created) supports inserting, modifying, and deleting a record via corresponding HTTP actions, and often you want to control that.

You can control the HTTP actions that the API supports (GET, POST, PATCH, DELETE, etc.) by using the `InsertAllowed`, `ModifyAllowed`, and `DeleteAllowed` properties of the API page.

## Using the read-only database replica

Dynamics 365 Business Central SaaS has a read-only replica of the production database active by default. When you have APIs that only need to read data, it's recommended to create APIs that work on this replica and not directly in the production database. This helps you maximize performance and avoid having high loads on the production database (where the users daily work).

To create an API that works on the read-only replica of the database (and so only supports reading data), you need to use the `DataAccessIntent` property of the API page as follows:

```
page 50202 PKTCustomerAPI
{
    PageType = API;
    Caption = 'customer';
    APIPublisher = 'PACKT';
    APIVersion = 'v1.0';
    APIGroup = 'customapi';
    EntityName = 'customer';
    EntitySetName = 'customers';
    SourceTable = Customer;
    ODataKeyFields = SystemId;
    DataAccessIntent = ReadOnly;
    Editable = false;
```

The `DataAccessIntent` property sets the data access intent of the page, and it can have the following values:

- `ReadWrite`: This permits accessing and modifying records (works on the production database).
- `ReadOnly`: This permits accessing records but not modifying them (works on the read-only database replica).

## Dynamics 365 APIs' operational limits

Remember that Dynamics 365 Business Central limits the number of simultaneous API calls that you can perform in a certain sliding window. If you have an external service that performs too many requests on a tenant, you could receive an HTTP 429 error (Too many requests reached):

```
{
  "error":
  {
    "code": "Application_TooManyRequests",
    "message": "Too many requests reached. Actual (101). Maximum (100)."
```

The purpose of this is mainly to avoid things such as **Denial-of-Service (DoS)** attacks and insufficient resources in the tenant.

The actual permitted maximum number of requests per minute is as follows:

- **Web service requests: 6,000** web service requests per **user** in the previous **5-minute sliding window**.
- **Concurrency limit for web service requests: 100** concurrently handled (5 processed, 95 queued) web service requests per **user**.

You can see some numbers about Dynamics 365 Business Central at the following link: <https://demiliani.com/2023/11/07/dynamics-365-business-central-is-my-on-premises-customer-ready-for-saas/>.

To avoid this situation, you should handle how you perform requests to a Dynamics 365 Business Central API endpoint and, if you receive this error, you should adopt something such as a **retry policy** on the API calls in your external application.

## Using OData bound actions

We can use Dynamics 365 Business Central APIs not only to perform CRUD operations on an entity but also to invoke standard business logic defined in the application (both custom and standard code).

We can trigger Dynamics 365 Business Central business logic associated with the current entity via APIs by using OData **bound actions**. Bound actions are essentially actions that you can invoke via HTTP calls for a selected entity.

Imagine that you want to have an API action to clone a selected customer. This action takes the context of the selected customer and then it creates a new customer record with the same field values (except the Name field).

To do that, we can add a bound action to the previously created Customer API.

To use OData V4 bound actions, you need to declare a procedure in the API page decorated with the `[ServiceEnabled]` attribute. The procedure is defined as follows:

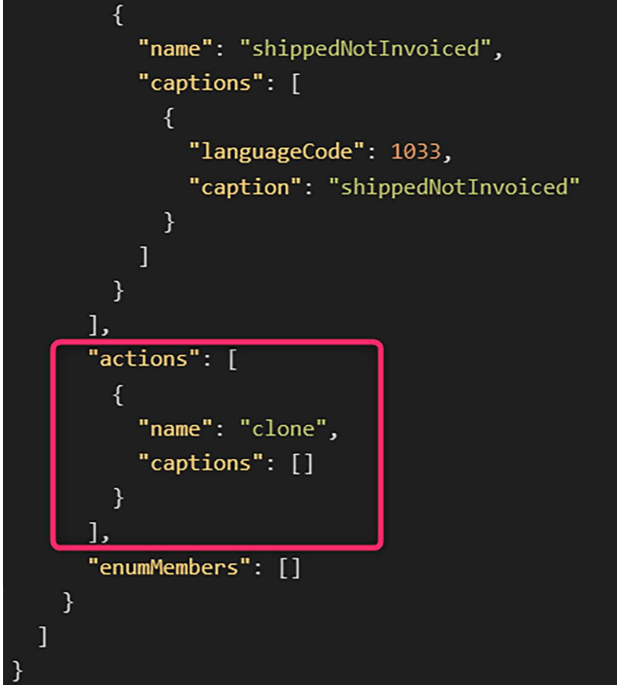
```
[ServiceEnabled]
procedure Clone (var ActionContext: WebServiceActionContext)
var
    Customer: Record Customer;
begin
    Customer.Init();
    Customer.TransferFields(Rec, false);
    Customer.Name := 'CUSTOMER CLONED VIA BOUND ACTION';
```

```
Customer.Insert(true);
    ActionContext.SetResultCode(WebServiceActionResultCode::Created);
end;
```

We can inspect the OData entity definition by sending a GET request to the following endpoint:

```
GET {{baseurl}}/api/packt/customapi/v1.0/entityDefinitions
```

From the response, you can see that now our Customer API exposes an action. Please note that the action name starts with a lowercase letter (clone) while our AL method has a capital letter (Clone):



```
{
  "name": "shippedNotInvoiced",
  "captions": [
    {
      "languageCode": 1033,
      "caption": "shippedNotInvoiced"
    }
  ]
},
  "actions": [
    {
      "name": "clone",
      "captions": []
    }
  ],
  "enumMembers": []
}
```

Figure 13.27: “clone” action exposed by the custom API

To call our bound action, we need to send a POST request to the following endpoint:

```
POST {{baseurl}}/api/packt/customapi/v1.0/companies({{companyid}})/
customers(71b117ef-e06c-ed11-81b5-6045bd8e591f)/Microsoft.NAV.clone
```

Here we’re calling the clone action by passing the SystemId of the customer record to clone (which, in this case, is Adatum Corporation).

The AL procedure is invoked and a Customer record has been created (that is, cloned by the customer with "No." = 10000 as the input):

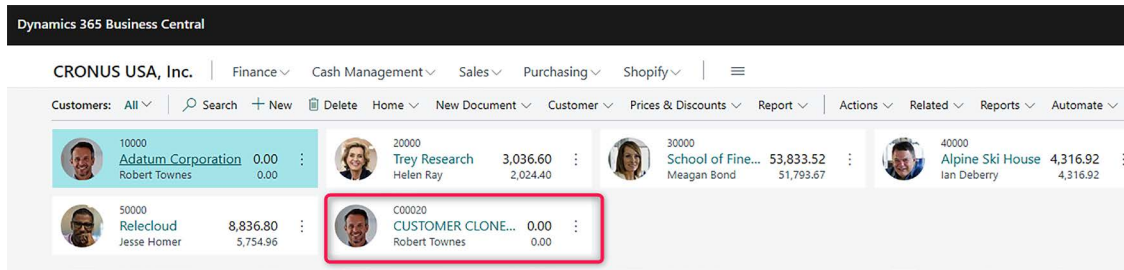


Figure 13.28: Testing the clone action

Here, you learned how to invoke actions linked to a particular record via an API.

In the next section, we'll see how to invoke custom AL code (business process not linked to an entity) via OData APIs.

## Using OData unbound actions

OData **unbound actions** are actions that you can invoke via HTTP requests and that are not bound to a particular entity. Unbound actions are useful to execute custom Dynamics 365 Business Central code from external applications.

To use **unbound actions** in Dynamics 365 Business Central, you need to:

1. Create a codeunit exposing your custom actions (procedures).
2. Register and publish the codeunit as a web service from Dynamics 365 Business Central.
3. Invoke the action via a POST OData call (passing the required parameters in the request body).

Imagine that we have an AL procedure that calculates the total amount of sales orders for a particular customer. The procedure is defined in a codeunit object as follows:

```
codeunit 50200 "PKT Unbound Actions"
{
    procedure GetSalesAmountForDay(CustomerNo: Code[20]) totalSales: Decimal;
    var
        SalesLine: Record "Sales Line";
    begin
        SalesLine.SetRange("Document Type", SalesLine."Document Type"::Order);
        SalesLine.SetRange("Sell-to Customer No.", CustomerNo);
        if SalesLine.FindSet() then
            repeat
                totalSales += SalesLine."Line Amount";
            until SalesLine.Next() = 0;
```

```
end;
}
```

When the extension containing that codeunit is published, we can open the **Web Services** page in Dynamics 365 Business Central and publish the codeunit as an OData endpoint as follows:

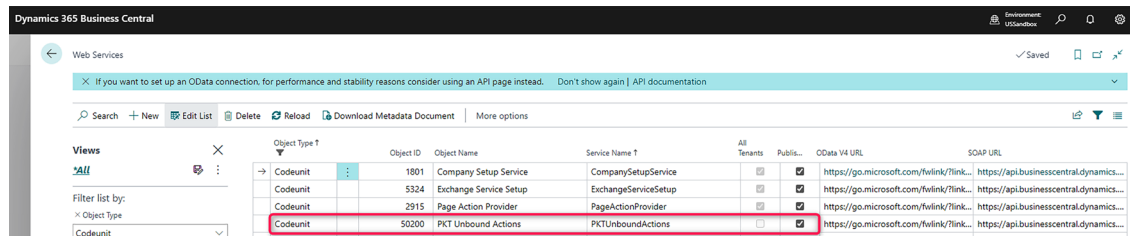


Figure 13.29: Publishing a codeunit as an OData endpoint

Here, you need to:

1. Set **Object Type** as **Codeunit**.
2. Set **Object ID** as the ID of the codeunit object that you want to expose.
3. Set **Service Name** as the name of the exposed service (don't use spaces). This will be part of your OData endpoint.
4. Set **Published** as **true**.

To call the published codeunit method as an OData unbound action, you need to send a POST request to the following endpoint:

```
POST {{baseurl}}/ODataV4/PKTUnboundActions_GetSalesAmountForCustomer?company=18
bb64bd-e06c-ed11-81b5-6045bd8e591f
```

Unbound actions must have the company ID as a parameter in the request URL. As previously said, if the procedure that you want to call requires parameters, you need to pass them in the JSON body of the request. Please note that the parameter names must be passed with a lowercase starting letter.

This is the complete call of our previously published action:

```
POST {{baseurl}}/ODataV4/PKTUnboundActions_GetSalesAmountForCustomer?company=18
bb64bd-e06c-ed11-81b5-6045bd8e591f
Authorization: Bearer {{token}}
Content-Type: application/json

{
  "customerNo": "10000"
}
```

This is the response, showing the total sales order amount for the selected customer:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true
Content-Encoding: gzip
Vary: Accept-Encoding
Server: Microsoft-HTTPAPI/2.0
Access-Control-Allow-Headers: Origin, X-Requested-With, Authorization
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
ms-correlation-x: e84a5f3b-c70b-0555-4a84-57920b3d4f80
mise-correlation-id: 53e953a6-8c4c-440e-8e80-d7b98c8e732a
OData-Version: 4.0
Access-Control-Expose-Headers: Date, Content-Length, Server, OData-Version, ms-correlation-x
request-id: 9bda8231-3569-40d4-864e-2f8d8ee49c7c
x-content-type-options: nosniff
Strict-Transport-Security: max-age=31536000; includeSubDomains
Date: Thu, 27 Jul 2023 10:08:01 GMT
Connection: close

{
  "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/US Sandbox/odata/v4/$metadata#Edm.Decimal",
  "value": 19052.9
}
```

Figure 13.30: Response when calling a codeunit with a POST request

By using **unbound actions**, you have the power to call every code you want from external applications, opening a wide range of integration scenarios.

In the next section, we'll see how to send requests in batches to an API endpoint.

## Using OData batch calls with Dynamics 365 Business Central APIs

Usually, when calling Dynamics 365 Business Central APIs, you perform a single operation at a time (you send an API request and you receive a response, then you send another API request, and so on).

OData also supports batch requests. With batch requests, you can combine multiple operations in a single OData request. The batch request is sent as a single POST operation to the OData \$batch endpoint, and the body of this request must contain a JSON with the set of operations to perform. This permits you to minimize the number of HTTP calls (requests) that you send to an API endpoint.

All APIs (standard and custom) in Dynamics 365 Business Central support batch requests. The \$batch endpoint is available at the following URLs:

- Standard APIs: `https://{baseurl}/api/v2.0/$batch`
- Custom APIs: `https://{baseurl}/api/[publisher]/[apigroup]/[version]/$batch`

When calling the `$batch` endpoint, you should always specify that you want a response in JSON format, so set the `Accept` header parameter as `application/json`:

```
POST {{baseurl}}/api/v2.0/$batch
Content-Type: application/json
Accept: application/json
```

When using batch requests, the request body must be a JSON object containing a `requests` array parameter of operations:

```
{
  "requests": []
}
```

Each request must have the following properties:

- `id` (mandatory): The unique identifier for the operation.
- `method` (mandatory): The HTTP method of the operation (GET, POST, PATCH, PUT, or DELETE).
- `url` (mandatory): The URL of the API to call (contains parameters like `$select`, `$filter`, etc.). This can be a relative path or an absolute path:
  - A relative path replaces the `$batch` part of the batch request URL.
  - An absolute path replaces the complete URL of the batch request.
- `headers` (optional): A list of the headers for the request.
- `body` (optional): The JSON body of the specific request.

As an example of how to compose a batch request, let's imagine inserting multiple journal lines via the standard Dynamics 365 Business Central `journalLines` API. We can send a batch request like the following:

```
POST {{baseurl}}/api/v2.0/$batch
Authorization: Bearer {{token}}
Content-Type: application/json
Accept: application/json

{
  "requests": [
    {
      "method": "POST",
      "id": "r1",
      "url": "companies({{companyId}})/journals({{journalId}})/
journalLines",
      "headers": {
        "Content-Type": "application/json"
      },

```

```

        "body": {
            "accountId": "{{accountId}}",
            "postingDate": "2023-07-27",
            "documentNumber": "JNL2023-07-01",
            "amount": -5000,
            "description": "Salary to Consultants"
        }
    },
    {
        "method": "POST",
        "id": "r2",
        "url": "companies({{companyId}})/journals({{journalId}})/
journalLines",
        "headers": {
            "Content-Type": "application/json"
        },
        "body": {
            "accountId": "{{accountId}}",
            "postingDate": "2023-07-27",
            "documentNumber": "JNL2023-07-02",
            "amount": -7500,
            "description": "Salary to Developers"
        }
    }
]
}

```

Here, we're sending two journal lines POST requests (named r1 and r2) with their details in a single batch call.

The API response is as follows:

```
"responses": [
  {
    "id": "r1",
    "status": 201,
    "headers": {
      "urls-rewritten-to-public": "false",
      "location": "https://api.businesscentral.dynamics.com/v2.0/USSandbox/api/v2.0/companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/journals(e4700afb-e06c-ed11-81b5-6045bd8e591f)/journalLines(5479fa3d-7e2c-ee11-bdf5-000d3a16ab42)",
      "content-type": "application/json; odata.metadata=minimal; odata.streaming=true",
      "odata-version": "4.0"
    },
    "body": {
      "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/USSandbox/api/v2.0/$metadata#companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/journals(e4700afb-e06c-ed11-81b5-6045bd8e591f)/journalLines/$entity",
      "@odata.etag": "W/\"JzIwOzE3NDASNzg5MzAzOTQ3NzI1MjVjMTswMDsn\"",
      "id": "5479fa3d-7e2c-ee11-bdf5-000d3a16ab42",
      "journalId": "e4700afb-e06c-ed11-81b5-6045bd8e591f",
      "journalDisplayName": "DEFAULT",
      "lineNumber": 10000,
      "accountType": "G_x002F_L_x0020_Account",
      "accountId": "2ab117ef-e06c-ed11-81b5-6045bd8e591f",
      "accountNumber": "10000",
      "postingDate": "2023-07-27",
      "documentNumber": "JNL2023-07-01",
      "externalDocumentNumber": "",
      "amount": -5000.00,
      "description": "Salary to Consultants",
      "comment": "",
      "taxCode": "",
      "balanceAccountType": "G_x002F_L_x0020_Account",
      "balancingAccountId": "00000000-0000-0000-0000-000000000000",
      "balancingAccountNumber": "",
      "lastModifiedDateTime": "2023-07-27T13:05:22.307Z"
    }
  },
  {
    "id": "r2",
    "status": 201,
    "headers": {
      "urls-rewritten-to-public": "false",
      "location": "https://api.businesscentral.dynamics.com/v2.0/USSandbox/api/v2.0/companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/journals(e4700afb-e06c-ed11-81b5-6045bd8e591f)/journalLines(5579fa3d-7e2c-ee11-bdf5-000d3a16ab42)",
      "content-type": "application/json; odata.metadata=minimal; odata.streaming=true",
      "odata-version": "4.0"
    },
    "body": {
      "@odata.context": "https://api.businesscentral.dynamics.com/v2.0/USSandbox/api/v2.0/$metadata#companies(18bb64bd-e06c-ed11-81b5-6045bd8e591f)/journals(e4700afb-e06c-ed11-81b5-6045bd8e591f)/journalLines/$entity",
      "@odata.etag": "W/\"JzIwOzE2MTM1ODc4NzU1MDAxNTU2OTI5MTswMDsn\"",
      "id": "5579fa3d-7e2c-ee11-bdf5-000d3a16ab42",
      "journalId": "e4700afb-e06c-ed11-81b5-6045bd8e591f",
      "journalDisplayName": "DEFAULT",
      "lineNumber": 20000,
```

Figure 13.31: API response to a POST request

As you can see, we have a single JSON response that contains an array of responses (a response for each operation we sent) with their status and their body. Each single result has the corresponding request ID identifier. Please remember that the order of the responses can be different from the order of the request you send.

When you send a batch of requests, some of them can be successfully executed and some of them can fail. There are scenarios where you also want to continue processing the other requests in the batch if a previous request fails, and there are other scenarios where you want to stop and completely roll back the entire batch.

If you want to continue to process the requests in the batch if a request fails, you need to set the `Prefer` header as follows:

```
POST {{baseurl}}/api/v2.0/$batch
Content-Type: application/json
Accept: application/json
Prefer: odata.continue-on-error
```

If you want a transactional behavior, you need to use the `OData-Isolation` header, as in the following:

```
POST {{baseurl}}/api/v2.0/$batch
Content-Type: application/json
Accept: application/json
Prefer: odata.continue-on-error
OData-Isolation: snapshot
```

In this way, you're using OData snapshot isolation. By also maintaining the `Prefer: odata.continue-on-error` parameter, you will obtain a full list of all the failing operations (instead of only the first one).

A batch API call can also contain different types of operations. You can, for example, send a single batch API request that contains operations for creating a customer, creating a vendor, creating a sales order for a new customer, and so on.

When sending multiple operations in a batch call, you need to remember that operations are always processed in parallel. But there are scenarios where this is not possible.

As an example, let's consider our previous batch example, where we create two journal lines in a single API request. Now imagine that we want also to post them, and the call to the posting method must be done in the same batch API request.

As you can imagine, we cannot post the journal lines before they are successfully inserted, so the posting operations must be **dependent** on the success of the previous insert requests. To specify that an operation in the batch should not start before the other operations, you can use the `dependsOn` property. The batch API request for this case will be like the following:

```
POST {{baseurl}}/api/v2.0/$batch
Authorization: Bearer {{token}}
Content-Type: application/json
```

Accept: application/json

```
{
  "requests": [
    {
      "method": "POST",
      "id": "r1",
      "url": "companies({{companyId}})/journals({{journalId}})/
journalLines",
      "headers": {
        "Content-Type": "application/json"
      },
      "body": {
        "accountId": "{{accountId}}",
        "postingDate": "2023-07-27",
        "documentNumber": "JNL2023-07-01",
        "amount": -5000,
        "description": "Salary to Consultants"
      }
    },
    {
      "method": "POST",
      "id": "r2",
      "url": "companies({{companyId}})/journals({{journalId}})/
journalLines",
      "headers": {
        "Content-Type": "application/json"
      },
      "body": {
        "accountId": "{{accountId}}",
        "postingDate": "2023-07-27",
        "documentNumber": "JNL2023-07-02",
        "amount": -7500,
        "description": "Salary to Developers"
      }
    },
    {
      "method": "POST",
      "id": "4",
      "dependsOn": ["r1", "r2"],
```

```

        "url": "companies({{companyId}})/journals({{journalId}})/
Microsoft.NAV.post",
        "headers": {
            "Content-Type": "application/json"
        },
        "body": { }
    }
]
}

```

We explored the possibilities for handling batch calls via OData APIs. Batch calls are useful when you want to optimize sending multiple requests to an API endpoint, and in some business scenarios, they can be extremely useful to use.

In the next section, we'll examine the concept of webhooks inside Dynamics 365 Business Central and we'll explore how to subscribe to notifications sent from a Dynamics 365 Business Central entity.

## Using Dynamics 365 Business Central webhooks

**Webhooks** are a way to create event-driven service integrations: instead of polling another system to check whether there are any changes in entities with webhooks, a client subscribes to events that will be pushed to it from the source system. Dynamics 365 Business Central supports webhooks, so a client can subscribe to a webhook notification (event) and will then automatically receive notifications if an entity in Dynamics 365 Business Central changes.

To use webhooks with Dynamics 365 Business Central, we need to perform the following steps:

1. A subscriber must register the webhook subscription with Dynamics 365 Business Central by making a POST request to the **Subscription** API and by passing a notification URL in the request body. The endpoint URL is as follows:

```

https://api.businesscentral.dynamics.com/v2.0/TENANTID/production/api/
v1.0/subscriptions

```

2. The request body to establish a subscription is the following (here, we're using the customers entity as an example):

```

{
    "notificationUrl": "YourApplicationUrl",
    "resource": "https://api.businesscentral.dynamics.com/v2.0/TENANTID/
production/api/v1.0/companies(COMPANYID)/customers",
    "clientState": "SomeSharedSecretForTheNotificationUrl"
}

```

3. Dynamics 365 Business Central returns a validation token to the subscriber.
4. The subscriber needs to return the validation token in the response body and provide the status code 200 (this is the mandatory handshake phase).

5. If Dynamics 365 Business Central receives the validation token in the response body, the subscription is registered and notifications will be sent to the notification URL.

When a subscription is established, the subscriber receives a notification for every update on the subscribed entity. Webhook subscriptions expire after 3 days if they are not renewed before that.

To renew a webhook subscription, a subscriber must send a PATCH request to the subscription endpoint (this request requires the handshake phase, too). The request endpoint to renew a webhook subscription is as follows:

```
https://api.businesscentral.dynamics.com/v2.0/TENANTID/production/api/v1.0/subscriptions('SUBSCRIPTIONID')
```



To renew a webhook subscription, you need to pass the `@odata.etag` tag of your previously established subscription as an If-Match block in the PATCH request header.

This is the HTTP response that you receive when the subscription is established:

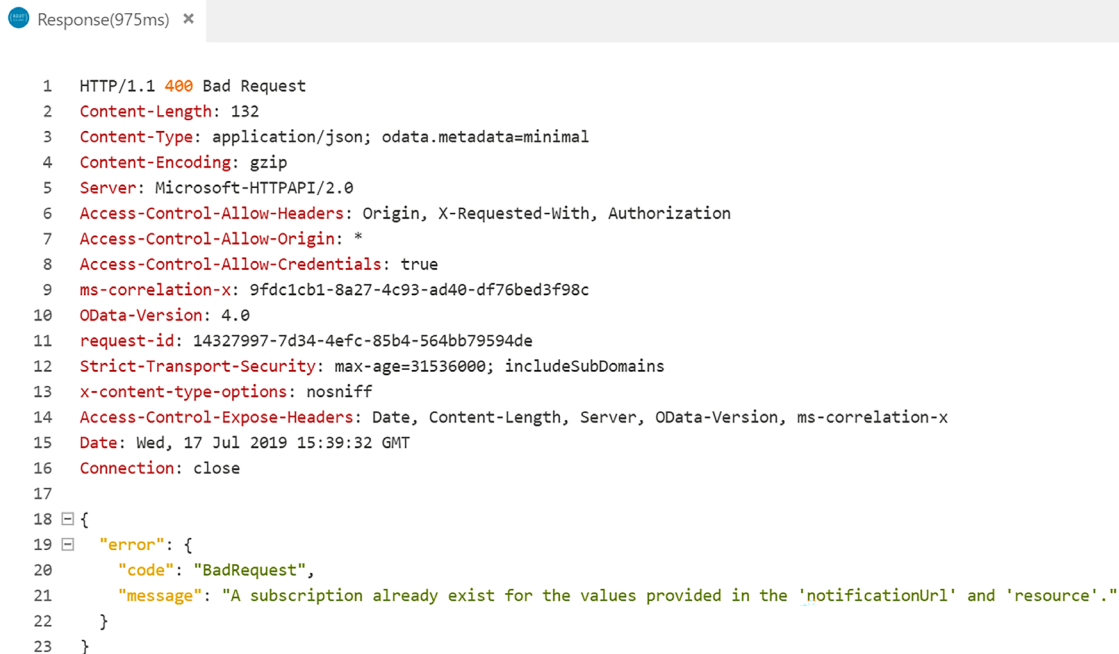
```

Response(1635ms) x
6  Server: Microsoft-HTTPAPI/2.0
7  Access-Control-Allow-Headers: Origin, X-Requested-With, Authorization
8  Access-Control-Allow-Origin: *
9  Access-Control-Allow-Credentials: true
10 ms-correlation-x: 1a6e52dd-fccc-4692-800f-f7e705b9ed9c
11 OData-Version: 4.0
12 request-id: 28fc1b91-d868-49f4-bc31-c733984438ec
13 Strict-Transport-Security: max-age=31536000; includeSubDomains
14 x-content-type-options: nosniff
15 Access-Control-Expose-Headers: Date, Content-Length, Server, OData-Version, ms-correlation-x
16 Date: Wed, 10 Jul 2019 15:45:42 GMT
17 Connection: close
18
19 {
20   "@odata.context": "https://api.businesscentral.dynamics.com/v1.0/194e87bd-73c6-43c6-95d7-1ca48985db5e/api/beta/$metadata#subscriptions/$entity",
21   "@odata.etag": "W/\"JzQ000lyemodod1UxawduR3NZaUp2aW9oVTFERU5vQV1JQXpLQi8rU3NyVzBsNmM9MTswMDsn\"",
22   "subscriptionId": "84765ab3c48c4bb0beeb789f936a8c71",
23   "notificationUrl": "https://d365bcwebhooks.azurewebsites.net/api/D365BCListener?code=MJ4Y2Pltvu0Op/4qz8GbDIQ56LtJqBPiV6Y1fP16cQvXIZL4oA2NRQ==&testerId=subscriber1",
24   "resource": "api/beta/companies(80d28ea6-02a3-4ec3-98f7-936c2000c7b3)/customers",
25   "userId": "10cb2537-0adf-4a69-97e5-894c60788667",
26   "lastModifiedDate": "2019-07-10T15:45:41Z",
27   "clientState": "",
28   "expirationDateTime": "2019-07-13T15:45:41Z"
29 }

```

Figure 13.32: HTTP response to a PATCH request

If you try to issue a subscription request again to the same endpoint where an active subscription has been established, you receive the following error:



```

Response(975ms) ✕
1  HTTP/1.1 400 Bad Request
2  Content-Length: 132
3  Content-Type: application/json; odata.metadata=minimal
4  Content-Encoding: gzip
5  Server: Microsoft-HTTPAPI/2.0
6  Access-Control-Allow-Headers: Origin, X-Requested-With, Authorization
7  Access-Control-Allow-Origin: *
8  Access-Control-Allow-Credentials: true
9  ms-correlation-x: 9fdc1cb1-8a27-4c93-ad40-df76bed3f98c
10 OData-Version: 4.0
11 request-id: 14327997-7d34-4efc-85b4-564bb79594de
12 Strict-Transport-Security: max-age=31536000; includeSubDomains
13 x-content-type-options: nosniff
14 Access-Control-Expose-Headers: Date, Content-Length, Server, OData-Version, ms-correlation-x
15 Date: Wed, 17 Jul 2019 15:39:32 GMT
16 Connection: close
17
18 {
19   "error": {
20     "code": "BadRequest",
21     "message": "A subscription already exist for the values provided in the 'notificationUrl' and 'resource'."
22   }
23 }

```

Figure 13.33: Error message to a PATCH request

When a subscription is established, a subscriber can receive notifications when the subscribed entities in Dynamics 365 Business Central are modified. This is an example of a notification sent to a subscriber (the notification is a JSON object that contains all of the modified entities):

```

{
  "value": [
    {
      "subscriptionId": "customers",
      "clientState": "someClientState",
      "expirationDateTime": "2019-07-20T07:52:31Z",
      "resource": "api/beta/companies(80d28ea6-02a3-4ec3-98f7-936c2000c7b3)/customers(26814998-936a-401c-81c1-0e848a64971d)",
      "changeType": "updated",
      "lastModifiedDateTime": "2019-07-19T12:54:20.467Z"
    }
  ],

```

```
{
  "subscriptionId": "webhookCustomersId",
  "clientState": "someClientState",
  "expirationDateTime": "2019-07-20T07:52:31Z",
  "resource": "api/beta/companies(80d28ea6-02a3-4ec3-98f7-
              936c2000c7b3)/customers(130bbd17-dbb9-4790-9b12-
              2b0e9c9d22c3)",
  "changeType": "created",
  "lastModifiedDateTime": "2019-07-19T12:54:26.057Z"
}
]
```

Webhooks are also exposed for custom objects in our extensions. A page with `PageType = API` will expose webhooks with the following limitations (these also apply to standard API pages):

- The page cannot have composite keys.
- The page cannot use temporary tables or system tables as `SourceTable`.

A subscription to a webhook can be deleted by sending a DELETE request to the `/subscriptions({id})` endpoint. Also, to delete a subscription, you need to send a request with the `If-Match` header containing `@odata.etag` of the subscription to delete.

For more information about Dynamics 365 Business Central webhooks, I recommend checking this post: <http://demiliani.com/2019/12/10/webhooks-with-dynamics-365-business-central/>.

## Summary

In this chapter, we gave an overview of how to use the OData stack (and RESTful APIs, in particular) for integration with Dynamics 365 Business Central. We saw how to handle authentication with Microsoft Entra ID, how to use standard APIs, how to create custom APIs, how to create applications that use Dynamics 365 Business Central APIs, and how to use advanced concepts such as batches, bound and unbound actions, and webhooks.

At the end of this chapter, you were given a complete overview of how to expose Dynamics 365 Business Central business logic and entities and how to handle integrations with external applications by using REST HTTP calls and webhooks. APIs are the recommended way for creating integrations between Dynamics 365 Business Central and external applications.

In the next chapter, we'll see how to use Azure Functions and other serverless services with Dynamics 365 Business Central extensions to improve your integrations.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>



# 14

## Extending Dynamics 365 Business Central with Azure Services

In the previous chapter, we saw how we can use Dynamics 365 Business Central APIs to create integrations. But sometimes, integrations between Dynamics 365 Business Central and an external system can be complex or can require the usage of different technologies or cloud services. Doing everything with AL is not always a good choice; outside the ERP box, there are lots of cloud services that you can use for your integrations or for extending the capabilities of the ERP system.

In this chapter, we'll talk about some Azure services you can use to enhance the standard product's capabilities and create more powerful integrations with external systems. Here, we'll talk about the following topics:

- Using Azure Functions with Dynamics 365 Business Central
- Using Azure Logic Apps for creating workflows with Dynamics 365 Business Central

By the end of this chapter, you will be able to extend the standard capabilities of AL by using serverless technologies and you will be able to create powerful low-code workflows for your ERP integrations.

### Overview of Azure Functions

**Azure Functions** is one of the serverless compute services offered by the Azure platform. With Azure Functions, you can deploy and run event-triggered code without the need to provision and manage the underlying infrastructure. Everything is managed by Azure under the hood for you.

Common use cases for Azure Functions in Dynamics 365 Business Central projects are:

- Integration with external systems
- Decoupling Dynamics 365 Business Central from an external system (so, acting like a middle tier)
- Executing timer-triggered processes externally
- Integration with Azure resources

- Execution of .NET code in the cloud
- Execution of third-party libraries (DLLs) in a fully serverless way

Functions made with Azure Functions can be created directly from the Azure portal or from developer tools like Visual Studio or Visual Studio Code. They can be developed in different supported languages like C#, F#, PowerShell, JavaScript, Java, Python, and so on.

Azure Functions has a lot of benefits for cloud projects. The functions have the following traits:

- They're scalable (can scale up and down according to the demand they receive)
- They can integrate different Azure services and react to different Azure-related events
- They're easy to deploy (you can deploy directly from the developer tools or from CI/CD with Azure DevOps or GitHub Actions)
- They're cost effective; they have different pricing plans and you can pay for what you use
- They permit you to extend the standard capabilities of AL and use complex .NET code or third-party libraries in a cloud environment

Azure Functions has three pricing plans:

- **Consumption plan:** This is a pay-as-you-go plan where you are billed based on the number of executions and the amount of time your function is running. This is the recommended plan in many situations where you have functions that are executed a few times in a day or with irregular patterns. In the Consumption plan, functions are executed in a shared environment and the instances can be set at zero if a function is not in use. The next time you execute that function, the runtime needs to restart it and you have a delay. This is called a **cold start**.
- **Premium plan:** This is a more advanced plan where you can define the minimum number of instances for a function that will be always running in order to avoid a cold start. This plan also has more capabilities, including advanced features like VNet integrations.
- **App Service plan:** This is a plan similar to the plan you use to deploy web apps on Azure: you select a set of resources (CPU, memory, and so on) and your functions will be executed on those resources. With this plan, you are responsible for scaling. The cost is fixed according to the set of resources you have acquired.

When talking about Azure Functions, we need to quickly explain the main components:

- **Azure Functions app:** This is the deployment unit for Azure Functions. A Functions app is the resource you deploy on Azure and it can contain multiple functions. All functions in the same Functions app share the same configurations (storage, runtime version, configurations, etc.).
- **Azure Functions runtime:** This is the component that runs your code.
- **Scale controller:** This is the component responsible for automatically scaling your function instances up and down according to the demand.
- **Application settings:** These are the configuration settings for your functions. When you deploy a function locally, you have settings in the `local.settings.json` file of your project. When you deploy the Functions app on Azure, settings are stored in the Azure portal.

- **Triggers:** These are the events that start the execution of a function. Triggers have associated data and you can have different types of triggers (HTTP triggers, triggers related to Azure resources, timer triggers, and so on).
- **Bindings:** These are a way to declaratively connect other resources to a function as parameters and you can have input and output bindings.

Azure Functions now supports two **execution modes**:

- **In-process:** The function code runs in the same process as the function host. In this execution mode (standard mode), there's a tight integration between the application and the runtime environment (the functions share APIs and binding types with the host process and it requires the same .NET version as the runtime).
- **Isolated:** The function code runs in a separate process. Isolated process mode allows you to use any compatible .NET version and select specific assembly versions without being constrained by the host process. This execution mode supports also features like dependency injection and middleware.

Isolated process mode is the recommended choice for the newest Azure Functions applications due to its flexibility, extensibility and support for modern development practices.

In the next section, we'll see how to create an Azure function, with the goal to create a new feature to use in Dynamics 365 Business Central. Then we will see how to deploy it on Azure.

## Creating functions with Azure Functions

In this section, we'll learn how to create an Azure function for generating QR codes with .NET and Visual Studio Code, and later, we will use this function in Dynamics 365 Business Central. We'll also show you how, in an Azure function, you can use standard, custom, and third-party libraries in your code. These libraries will be safely deployed in the cloud when you deploy the Azure Functions app.

In this section, we will use Visual Studio Code to develop our function because it's the standard development tool that you're using also for AL development. Please note that Visual Studio (full version) is another recommended tool for creating Azure functions. It will offer a lot of features and wizards for Azure function creation, debugging, and deployment.

For this example, we'll use a free third-party library called **QRCode** for generating QR codes. This library is available on NuGet here: <https://www.nuget.org/packages/QRCode/>.

To start using Visual Studio Code to create Azure functions, you need first to install some Microsoft extensions from the Visual Studio Code Marketplace. The extensions are the following:

- **Azure Functions:** This extension is needed to have all the features for creating, debugging, and deploying Azure functions from Visual Studio Code.
- **C#:** This extension will add the C# language support to Visual Studio Code.
- **Azure account:** This is needed to log in and manage your Azure subscriptions.

You also need to install the **Azure Functions Core tools**. More information can be found here: <https://learn.microsoft.com/en-us/azure/azure-functions/functions-run-local?tabs=windows%2Cportal%2Cv2%2Cbash&pivots=programming-language-csharp#install-the-azure-functions-core-tools>.

To start creating an Azure function, the first thing that we suggest you do is create a new Functions app in the Azure portal (we recommend creating the Functions app in the portal and then, when the app is created, you can deploy the code from your development tools). Open the Azure portal, select **Function App**, and then click on **Create**.

Here, you need to:

- Select the Azure subscription where you want to create the new Azure Functions app.
- Create a new resource group (we recommend you do this for every set of consistent resources you have) or select an existing one.
- Insert the Functions app name.
- Select **Code** as the deployment image.
- Select the runtime stack, in our case, .NET.
- Select the .NET runtime version. Here, you can select if you want to use the in-process or isolated execution mode (LTS = long-term support). In this example, I've created an in-process Functions app.

- Select the Azure region where you want to deploy the Functions app:

[Home](#) > [Function App](#) >

# Create Function App ...

**Basics**   Storage   Networking   Monitoring   Deployment   Tags   Review + create

Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.

## Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ

Microsoft Azure Germany 25a7119a2-17ac-405b-8385-c514a239a2

Resource Group \* ⓘ

sdqrcodegenerator

Create new

## Instance Details

Function App name \*

QRCodegenerator.azurewebsites.net

Do you want to deploy code or container image? \*

☒ Code   ☐ Container Image

Runtime stack \*

.NET

Version \*

6 (LTS)


Region \*



West Europe

Figure 14.1: Function app region configuration

In the **Hosting** section, select the Azure Functions hosting plan (**Consumption** in our case):

### Hosting

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#) 

- Hosting options and plans  
- ☒ Consumption (Serverless)  
Optimized for serverless and event-driven workloads.
  - ☐ Functions Premium  
Event based scaling and network isolation, ideal for workloads running continuously.
  - ☐ App service plan  
Fully isolated and dedicated environment suitable for workloads that need large SKUs or need to co-locate Web Apps and Functions.

*Figure 14.2: Functions app hosting configuration*


In the **Storage** panel, you can specify the name of the linked storage account for the Functions app. In the **Monitoring** panel, you can create or select an instance of the Application Insights service for collecting the telemetry and logs coming from your Azure Functions app.

In the **Deployment** panel, you can connect your Functions app to a GitHub account for continuous deployment of your function code:

[Home](#) > [Function App](#) >

# Create Function App ...

Basics   Storage   Networking   Monitoring   Deployment   Tags   Review + create

**Enable GitHub Actions to continuously deploy your app.** GitHub Actions is an automation framework that can build, test, and deploy your app whenever a new commit is made in your repository. If your code is in GitHub, choose your repository here and we will add a workflow file to automatically deploy your app to App Service. If your code is not in GitHub, go to the Deployment Center once the web app is created to set up your deployment. [Learn more](#) 

### GitHub Actions settings

Continuous deployment   ☒ Disable   ☐ Enable

### GitHub Actions details

Select your GitHub details, so Azure Web Apps can access your repository. You must have write access to your chosen repository to deploy with GitHub Actions.

GitHub account	demiliani
	<div>Change account ⓘ</div>
Organization	<div>Select organization ▼</div>
Repository	<div>Select repository ▼</div>
Branch	<div>Select branch ▼</div>

Figure 14.3: Functions app deployment configuration

When all settings are completed, click **Review + Create** and your Functions app will be deployed on your subscription:

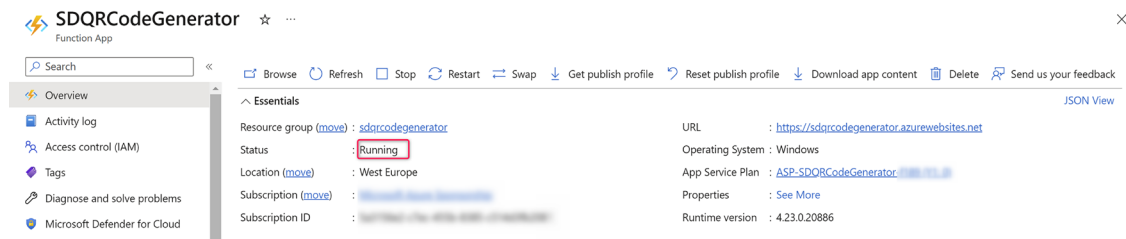


Figure 14.4: Functions app creation

Now, you're ready to start working on your Azure function code. Open Visual Studio Code and sign in to your Azure subscription (if not already done):

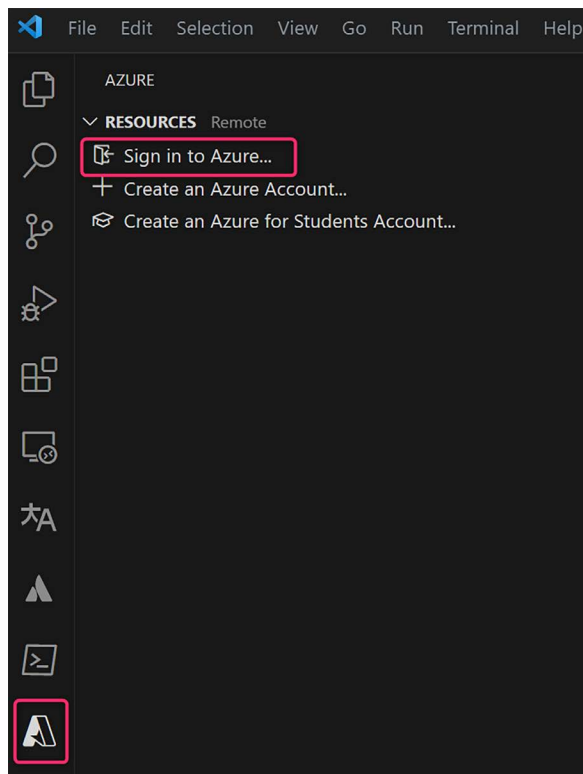
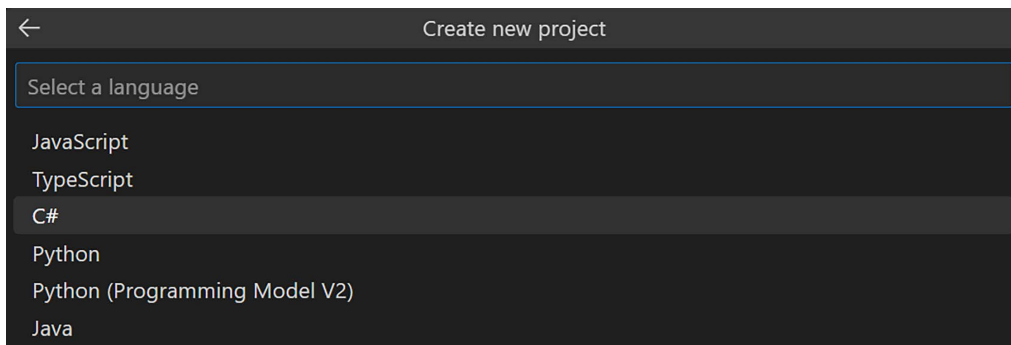


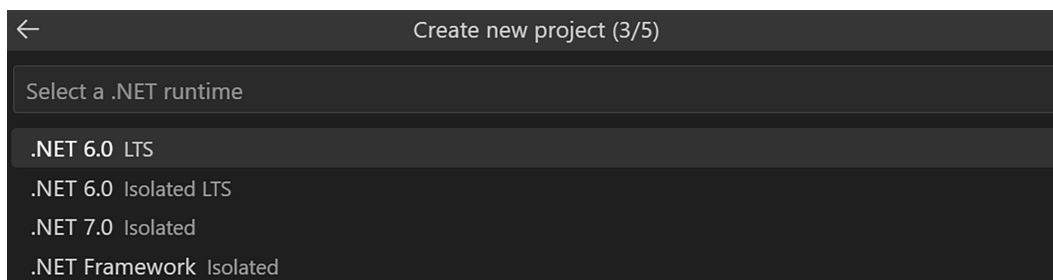
Figure 14.5: Signing in to Azure from Visual Studio Code

When logged in, open the Visual Studio Command Palette (*Ctrl + Shift + P*) and select the **Azure Functions: Create New Project** command. You will be prompted to specify a folder in which to locally save your Azure Functions project, then to select a language for developing the function code (select **C#**):



*Figure 14.6: Selecting a language for the project*

In the next prompt, select the Azure Functions runtime (**.NET 6.0 LTS** in our case):



*Figure 14.7: Selecting a runtime for the project*

Then, you need to select a template for your function. Select **HTTP trigger** (our function will be called via HTTP requests):

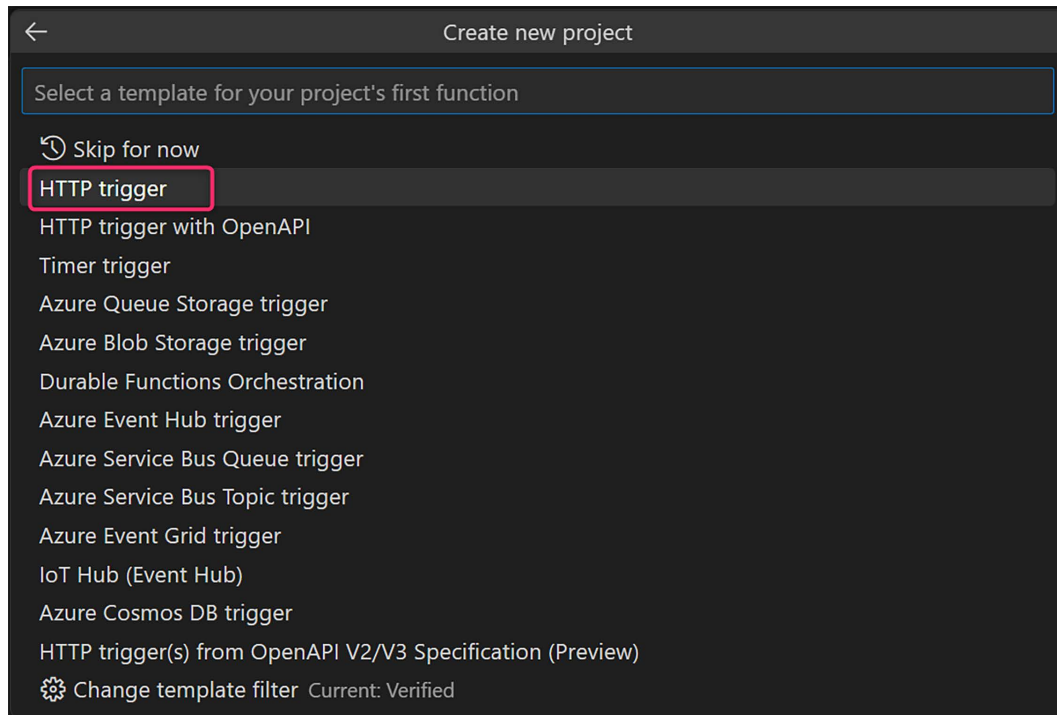


Figure 14.8: Selecting a project template

Then, specify a name for your function (like `QRGenerator`) and a namespace name (like `SD.QRGenerator`). For the **AccessRights** of your function, select **Function**, since we want to use function keys for authentication:

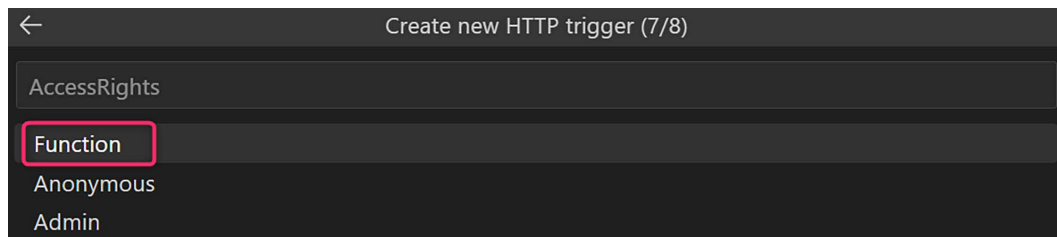
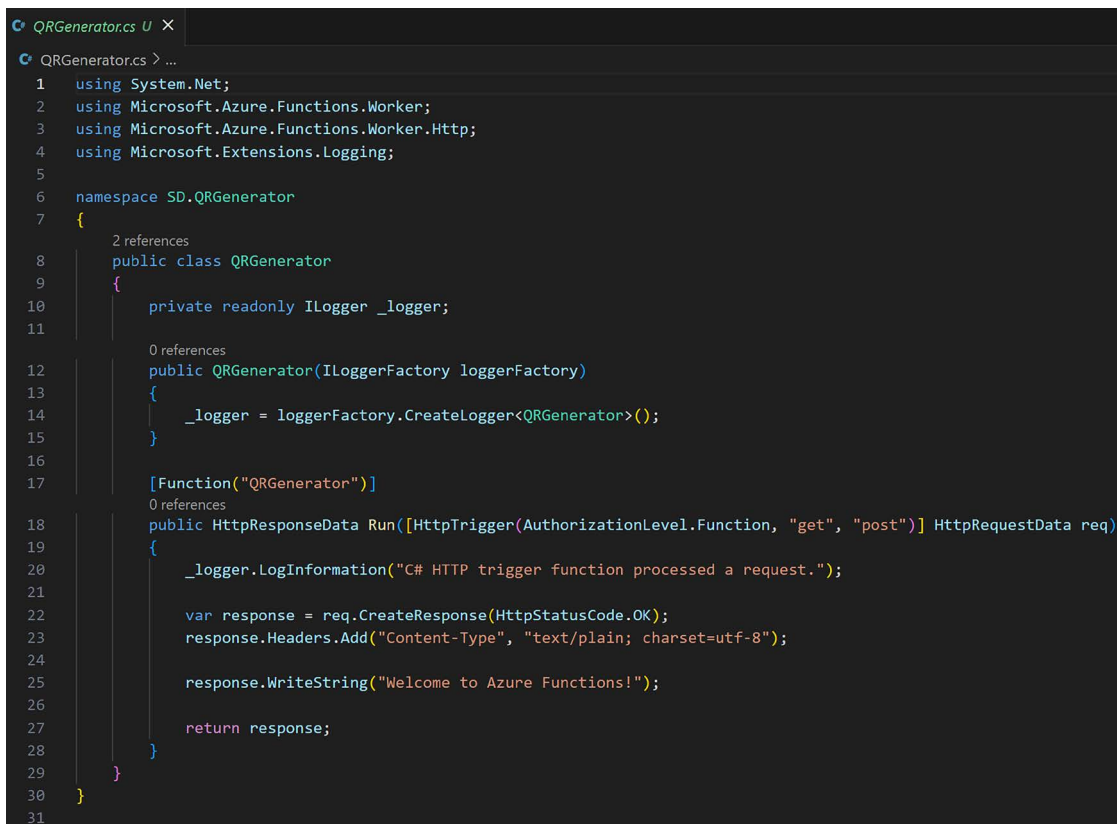


Figure 14.9: Specifying the function name, namespace, and access rights

A new Azure Functions project will be created and opened with a standard HTTP trigger template like the following:



```

1  using System.Net;
2  using Microsoft.Azure.Functions.Worker;
3  using Microsoft.Azure.Functions.Worker.Http;
4  using Microsoft.Extensions.Logging;
5
6  namespace SD.QRGenerator
7  {
8      2 references
8      public class QRGenerator
9      {
10         private readonly ILogger _logger;
11
12         0 references
12         public QRGenerator(ILoggerFactory loggerFactory)
13         {
14             _logger = loggerFactory.CreateLogger<QRGenerator>();
15         }
16
17         [Function("QRGenerator")]
18         0 references
18         public HttpResponseMessage Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")] HttpRequestData req)
19         {
20             _logger.LogInformation("C# HTTP trigger function processed a request.");
21
22             var response = req.CreateResponse(HttpStatusCode.OK);
23             response.Headers.Add("Content-Type", "text/plain; charset=utf-8");
24
25             response.WriteString("Welcome to Azure Functions!");
26
27             return response;
28         }
29     }
30 }
31

```

Figure 14.10: Project created in line with specified configuration

For our Azure Functions project, we need to add a reference to a third-party .NET assembly from NuGet. To do that, from Visual Studio Code, click on the terminal and execute the following command:

```
dotnet add package QRCode --version 1.4.3
```

The package reference will be added to the project.

Now, replace the standard template with the following C# code:

```

using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

```

```

using QRCoder;

namespace SD.QRGenerator
{
    public static class QRGenerator
    {
        [FunctionName("QRGenerator")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route =
null)] HttpRequest req,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a
request.");

            string url = req.Query["url"];

            string requestBody = await new StreamReader(req.Body).
ReadToEndAsync();
            dynamic data = JsonConvert.DeserializeObject(requestBody);
            url = url ?? data?.url;
            if (string.IsNullOrEmpty(url))
            {
                return new BadRequestResult();
            }
            var isAbsoluteUrl = Uri.TryCreate(url, UriKind.Absolute, out Uri
resultUrl);
            if (!isAbsoluteUrl)
            {
                return new BadRequestResult();
            }

            var generator = new Uri(resultUrl.AbsoluteUri);
            var payload = generator.ToString();

            using (var qrGenerator = new QRCodeGenerator())
            {
                var qrCodeData = qrGenerator.CreateQrCode(payload,
QRCodeGenerator.ECCLLevel.Q);
                var qrCode = new PngByteQRCode(qrCodeData);
                var qrCodePng = qrCode.GetGraphic(20);
                return new FileContentResult(qrCodePng, "image/png");
            }
        }
    }
}

```

```

    }
  }
}

```

Explaining the C# code in detail is beyond the scope of this book. This code essentially receives a parameter called `url` via an HTTP GET or POST request, verifies that the URL is valid and (if so) calls the third-party library (**QRCode**) to generate a QR code image containing that URL. Then, the generated image is returned to the caller as a response (binary stream with image/PNG content).

You can now press the **Run and Debug** button in Visual Studio Code (*Ctrl + Shift + D*) to test your Azure functions locally (with a local URL like the following): `http://localhost:7071/api/QRGenerator?url=http://www.packtpub.com`

To deploy your Azure function to the previously created Functions app on Azure, open the Visual Studio Code Command Palette and select the **Azure Functions: Deploy to Function App...** command:

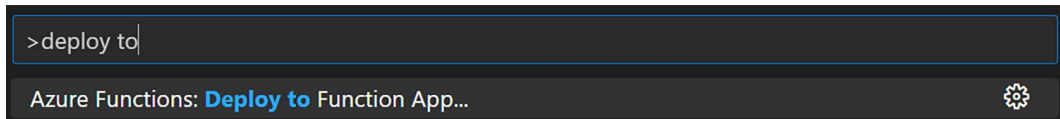


Figure 14.11: Deploying a function to the Functions app

When prompted, select your Azure subscription and the Functions app where you want to deploy the code.

The **QRGenerator** function will be deployed on the selected Functions app. By selecting the function name, you can also grab the URL for calling the function:

Home > SDQRCodeGenerator >

**QRGenerator** ...  
Function

Search « ✓ Enable ⏸ Disable 🗑 Delete **📄 Get Function Url** ↻ Refresh

{fx} Overview

Developer

Code + Test

Integration

Monitor

Function Keys

Get Function Url

default (fu... ▼

<https://sdqrcodegenerator.azurewebsites.net/api/QRGener...> 📄

OK

Subscription (move) : [Microsoft Azure Subscription](#)

Subscription ID : [5a1156a2-c76e-405b-8081-c714d78d2081](#)

Function app : [SDQRCodeGenerator](#)

Figure 14.12: Function deployed in a Functions app

We've used **Function** as the authorization level for our HTTP trigger function, so to successfully call it, we need to pass the function key in the URL (if we're using a GET method) or in the function body (if we're using a POST method).

To test our Azure functions deployed on Azure, we can, for example, send a POST request with the following JSON body (and with the function key passed in the request header via the `x-functions-key` parameter):

```
POST https://sdqrcodegenerator.azurewebsites.net/api/QRGenerator
x-functions-key: YOURFUNCTIONKEY

{
  "url": "http://www.packtpub.com"
}
```

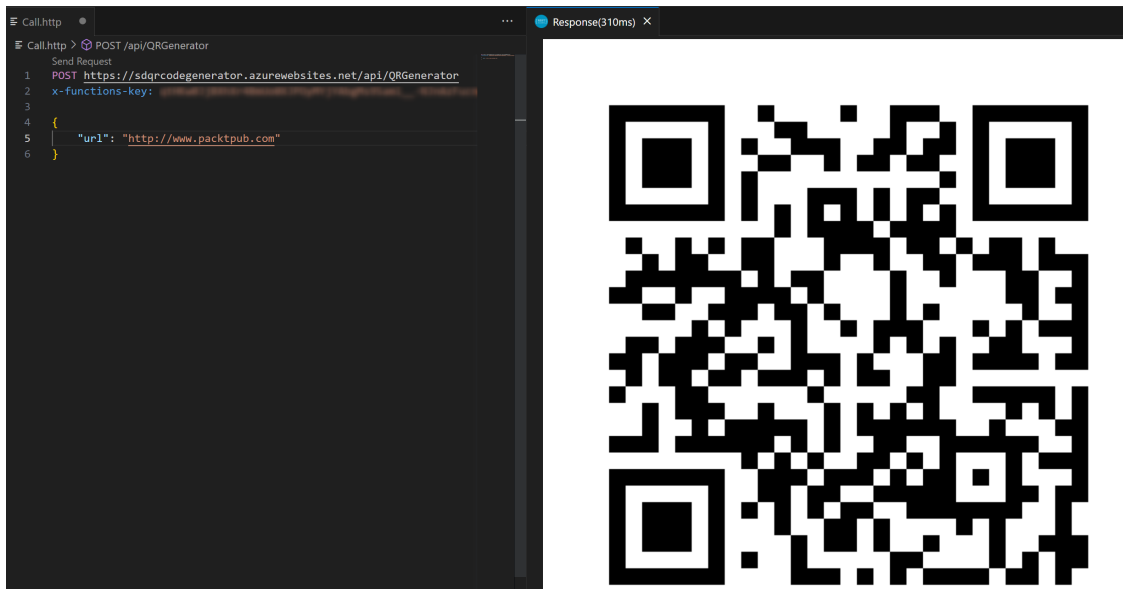


Figure 14.13: Testing a function with a POST request

The function key can be retrieved by going to the Functions app via the Azure portal and then selecting the **App keys** blade:

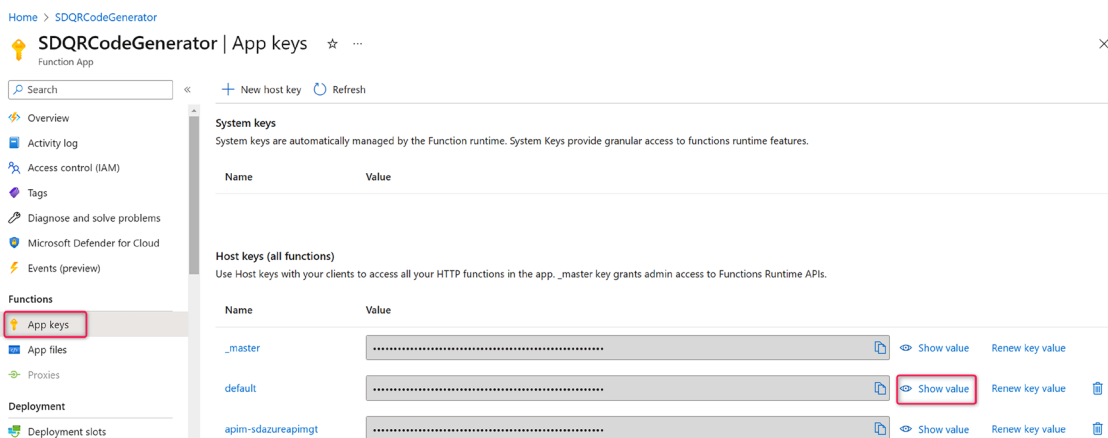


Figure 14.14: Retrieving the function key from the Functions app

We're now ready to use our Azure function from AL code in Dynamics 365 Business Central.

## Using Azure Functions from AL

Now, we want to use our Azure Functions app to generate QR codes from Dynamics 365 Business Central. In this example, imagine that we want to add an action in a Dynamics 365 Business Central extension to generate a QR code from the Customer Card, containing some details of the selected customer record (for example, the customer's **Home Page** field).

To do that, in this extension, we create a new pageextension object adding an action called Generate Customer QR Code to the Customer Card. Then, in the action code, we call a codeunit object containing the code for calling the Azure Functions.

The pageextension object code is the following:

```
pageextension 50100 SDCustomerCardExt extends "Customer Card"
{
    actions
    {
        addlast(processing)
        {
            action(SDQRCode)
            {
                ApplicationArea = All;
                Caption = 'Generate Customer QR Code';
                Image = Barcode;

                trigger OnAction()
                var
```

```

        AFDemo: Codeunit "SD Azure Functions Demo";
    begin
        AFDemo.GenerateQRCode(Rec."No.", Rec."Home Page");
    end;
}
}
addlast(Promoted)
{
    actionref(SDQRCodeRef; SDQRCode)
    {
    }
}
}
}
}

```

The codeunit code contains the code for calling the Azure function. It uses the **Azure Functions AL** module defined in the **System** application that permits you to easily handle authentication and send GET or POST requests to an Azure function. The code is the following:

```

codeunit 50101 "SD Azure Functions Demo"
{
    procedure GenerateQRCode(CustomerNo: Code[20]; URL: Text)
    var
        funcUrl: Label 'https://sdqrcodegenerator.azurewebsites.net/api/
QRGenerator';
        AzureFunction: Codeunit "Azure Functions";
        AzureFunctionResponse: Codeunit "Azure Functions Response";
        AzureFunctionAuthentication: Codeunit "Azure Functions Authentication";
        IAzurefunctionAuthentication: Interface "Azure Functions
Authentication";
        FunctionCode, Body, Filename : Text;
        ResultInStream: InStream;
    begin
        FunctionCode := YOUR FUNCTION CODE HERE';
    if URL <> '' then begin
        IAzurefunctionAuthentication := AzureFunctionAuthentication.
CreateCodeAuth(funcUrl, FunctionCode);
        Body := '{ "url": "" + URL + ""}';
        AzureFunctionResponse := AzureFunction.
SendPostRequest(IAzurefunctionAuthentication, Body, 'application/json');
    end;
}

```

```
        if AzureFunctionResponse.IsSuccessful() then begin
            Filename := CustomerNo + '.jpg';
            AzureFunctionResponse.GetResultAsStream(ResultInStream);
            DownloadFromStream(ResultInStream, 'QR CODE', '', '',
Filename);
        end;
    end;
end;
}
```

As you can see, here we authenticate to the Functions app by using the `CreateCodeAuth` method of the `AzureFunctionAuthentication` codeunit. This method returns an `AzureFunctionAuthentication` interface that we need to send to the next request together with the request body.

To handle the Azure function POST request, we create a JSON body and use the `SendPostRequest` method:

```
procedure SendPostRequest(AzureFunctionsAuthentication: Interface "Azure
Functions Authentication", Body: Text, ContentTypeHeader: Text): Codeunit
"Azure Functions Response"
```

This method returns an `Azure Functions Response` codeunit object containing the response of the call. If the response is successfully executed, we retrieve the result as a stream and we download it to the client.

When we deploy this extension, this is the new action that we have on the **Customer Card** page:

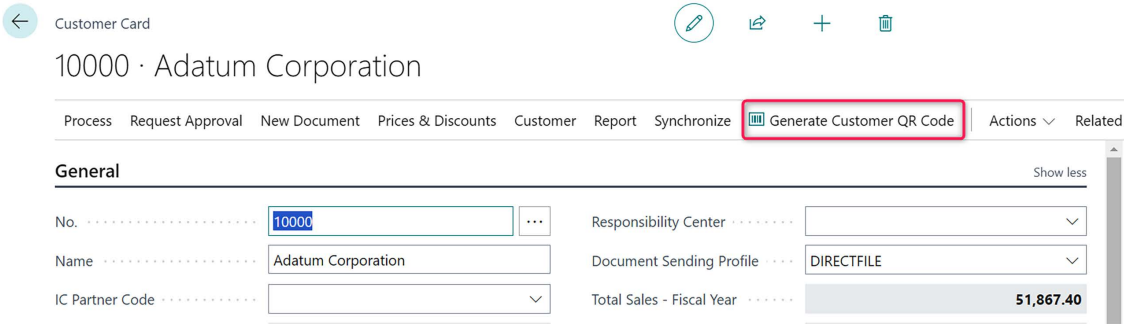


Figure 14.15: New action after deploying the extension

If the selected Customer record has a valid value in the **Home Page** field, a QR code is generated by calling the Azure function, and its content (JPG image) is downloaded to the client:

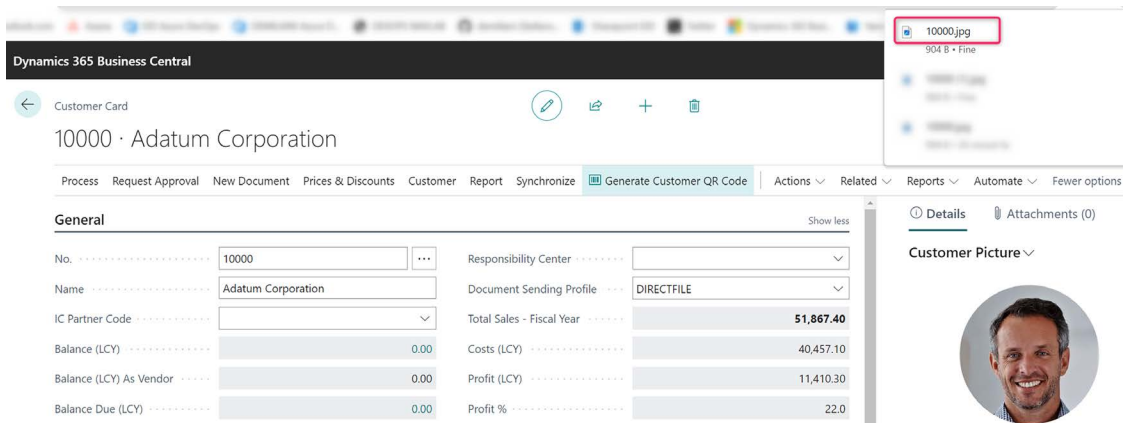


Figure 14.16: Testing the new action

With Azure Functions, you can easily add complex integrations and functionalities to your apps in a fully serverless and cost-effective way.

Azure Functions is a full-code solution for creating complex integrations in a serverless way. In the next section, we'll see how to create integrations and workflows for Dynamics 365 Business Central in a low-code way by using Azure technologies.

## Overview of Azure Logic Apps

**Azure Logic Apps** is a cloud platform for creating and orchestrating cloud workflows and integrating applications with a low-code approach.

Azure Logic Apps is a super-set of Power Automate (Microsoft's tool in Power Platform for workflow and task automation) and it's a great service if you want to create integrations between Dynamics 365 Business Central and other applications using low-code technologies and have scalability.

To design workflows, Azure Logic Apps provide a user-friendly designer (the same as Power Automate) via the Azure portal or Visual Studio Code/Visual Studio. You can read more about Azure Logic Apps here: <https://learn.microsoft.com/en-us/azure/logic-apps/logic-apps-overview>.

While Power Automate runs on Microsoft 365 and is bound to a user who creates the workflow, Azure Logic Apps app is a service that runs on an Azure subscription with two main tiers:

- **Consumption:** This is the pay-as-you-go tier. You pay per execution of actions and there's no difference between the Premium or Standard connector (you can use both). An Azure Logic App deployed in the Consumption tier can contain one workflow.

- **Standard:** This is the compute-based offer, where workflows can be deployed on an App Service plan or also hosted in Docker containers. A Logic Apps Standard plan can contain multiple stateful or stateless workflows.

When creating low-code workflows for Dynamics 365 Business Central, you need to remember that these two low-code workflow orchestration platforms have lots of similarities (Power Automate relies on Logic Apps) but also many important differences. The most important things to remember are listed below.

For Power Automate, consider the following:

- Built and maintained from `flow.microsoft.com`. Only users with a Power Automate license can access it.
- Flows are bound to a user and usable by others only after sharing it.
- Two built-in roles (**Owners** and **Run-only**). If you remove all owners of a flow, it will be removed as well.
- The location of the flow is determined by the location of the Power Platform environment (this means that your flow runs in the Azure Region where your Dataverse environment is located).
- Not possible to filter who can trigger the flow.
- It has basic logging (notifications for flow owners and flow monitors).
- The ALM process (DevOps) is managed via Power Platform solutions.
- It has a fixed cost per user (free and Premium tier). Some connectors require a Premium license.

In comparison, consider these aspects of Azure Logic Apps:

- Built and maintained through the Azure portal or Visual Studio Code/Visual Studio.
- Flows are bound to a resource group.
- Full RBAC capabilities of Azure (resource group permissions, Logic App Contributor, and Logic App Operator).
- You can choose the region where the app will run, which guarantees full scalability.
- Has the full set of Azure security features. This means that it can facilitate managed identities, Azure API Management integration, and configuring restrictions on who can trigger the flow (via **Workflow Settings**).
- Advanced logging and alerting, thanks to Azure Monitor.
- Possibility to use .NET assemblies in actions.
- Full DevOps (ARM templates) and code view.
- Cost based on consumption (billed per executed action) or fixed.

You can find more details about Azure Logic Apps, a comparison with Power Automate, and the benefits it has for Dynamics 365 Business Central projects by following this link on my personal website: <https://demiliani.com/tag/azure-logic-apps/>.

Azure Logic Apps is often the recommended solution for creating low-code workflows for integrating applications with Dynamics 365 Business Central, especially if you want to control performances, scalability, and reliability, and you don't want to have workflows linked to a particular user.

In the next section, you'll learn how to create workflows for Dynamics 365 Business Central by using Azure Logic Apps.

## Creating workflows with Azure Logic Apps

The easiest way to start creating a workflow with Azure Logic Apps is by going to the Azure portal and then selecting **Logic Apps**. Then, click on **Add**.

Here, you first need to select the Azure subscription and the resource group (as every Azure resource). Then, give the name to the Logic Apps app, select the Azure region where you want to deploy your workflow, and select the deployment type (**Workflow** or **Docker Container**). Then, select if you want to enable log analytics (we suggest enabling it in production workflows) and select the Logic Apps plan (**Standard** or **Consumption**; we selected **Consumption** here). You can also specify if you want to enable zone redundancy or not:

[Home](#) > [Logic apps](#) >

# Create Logic App ...

Basics

Tags

Review + create

Create a logic app, which lets you group workflows as a logical unit for easier management, deployment and sharing of resources. Workflows let you connect your business-critical apps and services with Azure Logic Apps, automating your workflows without writing a single line of code.

### Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ

Microsoft Azure Subscription: 72d7766c-7766-4776-8776-776677667766 ▼

Resource Group \* ⓘ

(New) pktdemologicapprg ▼

Create new

### Instance Details

Logic App name \*

PACKTDEMO ✓

Region \*

West Europe ▼

Enable log analytics \*

☐ Yes ☒ No

### Plan

The plan type you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type \*

☐ **Standard:** Best for enterprise-level, serverless applications, with event-based scaling and networking isolation.

☒ **Consumption:** Best for entry-level. Pay only as much as your workflow runs.

☐ Looking for the classic consumption create experience? [Click here](#)

### Zone redundancy (preview)

Set up your Consumption logic app to use availability zones in Azure regions that support zone redundancy. This option is available only when you create and deploy your logic app. Eventually, all Consumption logic apps in zone supported regions will enable availability zones by default. [Learn more](#)

Zone redundancy

☐ **Enabled:** Your Consumption logic app uses availability zone.

☒ **Disabled:** Your Consumption logic app doesn't use availability zones.

Review + create

< Previous

Next : Tags >

Figure 14.17: Configuring the Logic Apps resource

If you click the **Review + create** button, the Azure Logic Apps resource will be deployed.

Now, by going to the **Logic Apps Designer** of the newly created app, you can start authoring your workflow (you can start from predefined templates or from a blank canvas as we love to do):

[Home](#) > [Microsoft.Web-LogicAppConsumption-Portal-](#) | [Overview](#) > [PACKTDEMO](#) >

## Logic Apps Designer ...

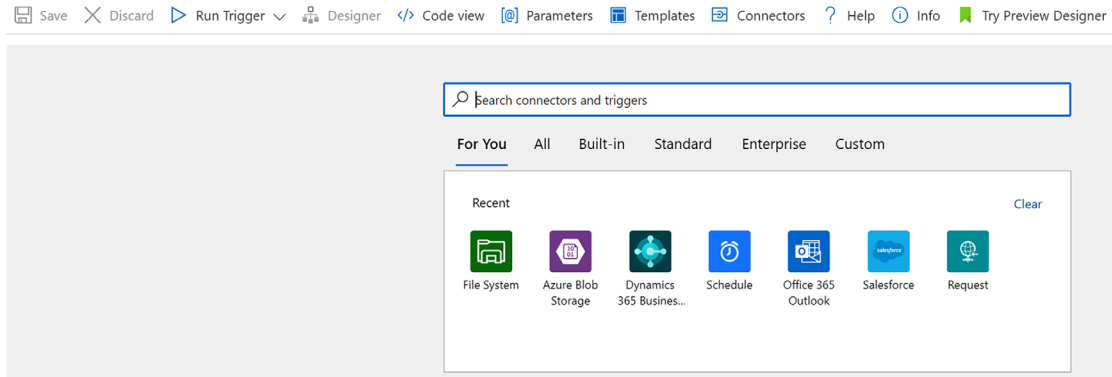


Figure 14.18: Logic Apps Designer

From here, you can use all the available connectors (standard and custom) you have in Power Automate (and some more) to create your workflows. In Azure Logic Apps, there's no difference between Standard or Premium connectors. The authoring workflow experience and rules are the same as you have on Power Automate.

When designing a workflow, you start from a trigger (the event that starts the workflow) and then you can create actions and conditions accordingly. You can create workflows that start from an HTTP request, from a recurrence, from events occurring on Azure resources, or from something happening on other applications.

Here is an example of a workflow that starts when a business event occurs in Dynamics 365 Business Central (**Purchase order released** event) and then retrieves the order details and saves a file in a Blob Storage account:

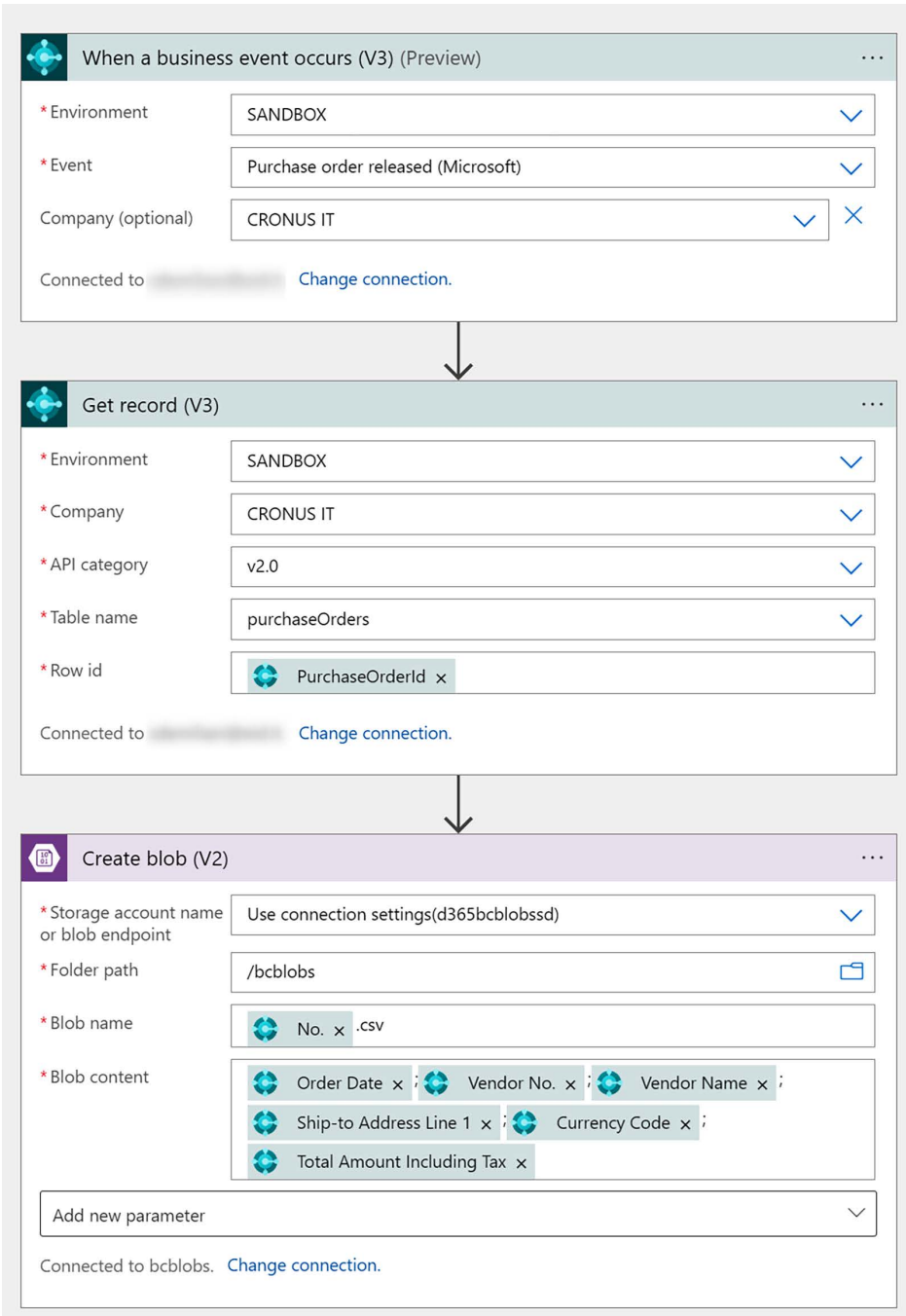


Figure 14.19: Blob Storage workflow

Please note that environment and Company can also be passed as parameters between actions (every action in the Business Central connector passes these two parameters to the other actions).

When you save the workflow, it will be listed as active and running. You can monitor it from the Azure portal or Azure Application Insights and you can disable it when not in use.

Azure Logic Apps usage in Dynamics 365 Business Central projects opens the possibility to use low-code approaches for a wide range of possible integrations, like:

- Integrations with Azure resources
- Integration with external databases (like Azure SQL or others)
- Integration with on-premises resources (like files, on-premises SQL databases, Oracle, etc.).
- Integrations with Dataverse
- Integrations with different types of other external applications (by using a large set of pre-defined connectors)

## Summary

In this chapter, you learned how to extend Dynamics 365 Business Central by using some serverless services offered by the Azure platform. In particular, you learned how to create Azure functions and how to use them from AL code for extending the capabilities of the ERP and for using (or reusing) .NET assemblies (standard, custom, or third-party ones). Then, you saw how to create low-code workflows for integrating Dynamics 365 Business Central by using Azure Logic Apps and you learned the main differences compared to Power Automate.

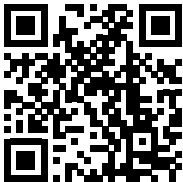
Now, you have the full power of Azure in your hands for creating no-limits and reliable functionalities and integrations for Dynamics 365 Business Central.

In the next chapter, we'll focus the attention on applying DevOps principles for Dynamics 365 Business Central extension's development and we'll see how we can automate CI/CD tasks in our AL development process.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>



# 15

## DevOps for Dynamics 365 Business Central

In the previous chapter, we saw how we can extend Dynamics 365 Business Central functionalities and improve integrations with external applications by using Azure cloud services, like Azure Functions and Azure Logic Apps.

In this chapter, we'll focus our attention on the Dynamics 365 Business Central extension development process. Developing extensions for Dynamics 365 Business Central professionally requires rules, practices, and tools in order to deliver better software and improve development team efficiency. This is what we call "DevOps."

To help maintain a professional DevOps process when developing extensions for Dynamics 365 Business Central, Microsoft created the **AL-Go for GitHub** project.

In this chapter, we will cover the following topics:

- What AL-Go for GitHub is
- Source code management and branching strategies for repositories
- Creating an extension project with AL-Go for GitHub
- Handling dependencies between apps in AL-Go for GitHub
- Creating CI/CD pipelines
- Creating automatic releases of apps

### AL-Go for GitHub: an introduction

**AL-Go for GitHub** (<https://github.com/microsoft/AL-Go>) is a set of GitHub templates and actions that can be used to set up and maintain professional DevOps processes for your Dynamics 365 Business Central AL extension's projects (per-tenant extensions and AppSource extensions).

The project is officially maintained by Microsoft, and you can follow the roadmap at the following link: <https://github.com/orgs/microsoft/projects/521/views/1>.

With AL-Go for GitHub, Microsoft aims to provide partners and customers with a set of automated tools and tasks to apply DevOps techniques to AL development. AL-Go for GitHub is open source and free for everyone (the unique requirement is to have a GitHub account), supports the development of per-tenant extensions (PTEs) and AppSource apps, is easy to get started and maintain, and does not require prior knowledge of things like Docker, Powershell, and YAML. With *AL-Go for GitHub*, DevOps becomes a tool and not an investment area.

As previously said, AL-Go for GitHub relies on GitHub, the code hosting platform owned by Microsoft, who heavily invests in it. As of June 2022, GitHub reported having over 83 million developers and more than 200 million repositories, including at least 28 million public repositories. It is the largest source code host as of November 2021.

To work with GitHub, you need to have an account on it. There are three types of accounts on GitHub:

- **Personal accounts:** Every person who uses GitHub.com signs into a personal account. Each personal account uses either GitHub Free or GitHub Pro. All personal accounts can own an unlimited number of public and private repositories, with an unlimited number of collaborators on those repositories. If you use GitHub Free, private repositories owned by your personal account have a limited feature set. You can upgrade to GitHub Pro to get a full feature set for private repositories.
- **Organization accounts:** Organizations are shared accounts where an unlimited number of people can collaborate on many projects at once. Like personal accounts, organizations can own resources such as repositories, packages, and projects. However, you cannot sign into an organization. Instead, each person signs into their own personal account, and any actions the person takes on organization resources are attributed to their personal account. Each personal account can be a member of multiple organizations. The personal accounts within an organization can be given different roles in the organization, which grant different levels of access to the organization and its data. All members can collaborate with each other in repositories and projects, but only organization owners and security managers can manage the settings for the organization and control access to the organization's data, with sophisticated security and administrative features. All organizations can own an unlimited number of public and private repositories. You can sign up organizations for free, with GitHub Free, which includes limited features on private repositories. To get the full feature set on private repositories and additional features at the organization level, including SAML single sign-on and improved support coverage, you can upgrade to GitHub Team or GitHub Enterprise Cloud.
- **Enterprise accounts:** GitHub Enterprise Cloud and GitHub Enterprise Server include enterprise accounts, which allow administrators to centrally manage policy and billing for multiple organizations and enable InnerSourcing between organizations.

GitHub offers free and paid plans for hosting your code. Plan details can be found at the following link: <https://github.com/pricing>.

The recommendation is to use an **organizational account** when:

- The owner of an organization owns the code.
- You are an ISV implementing AppSource apps (you should place the codebase in one or more repositories in one organization).
- You are a VAR implementing per-tenant extensions (you should place the codebase in in an organization owned by the customer with the partner as a collaborator). A free account is likely sufficient for customer organization accounts.

Organizational accounts can have shared runners (runners defined in an organization can be used by all repositories) and shared secrets (Teams or Enterprise accounts can create secrets in an organization and share them with repositories). With organizational accounts, billing goes to the organization instead of your personal account.

Repositories should be configured as private unless you want to create an open-source app.

AL-Go for GitHub has two main **templates** for creating AL extensions:

- <https://github.com/microsoft/AL-Go-PTE> or <https://aka.ms/algopte> is the GitHub repository template for **Per Tenant Extensions**.
- <https://github.com/microsoft/AL-Go-AppSource> or <https://aka.ms/algoappsource> is the GitHub repository template for **AppSource apps**.

These repositories are the starting point for creating apps with **AL-Go for GitHub**.

AL-Go for GitHub supports single-project and multi-project repositories in the following ways:

- Every project can contain multiple apps. These apps have the following traits:
  - All apps in a project need to be able to be installed together.
  - Apps within a project can have inter-dependencies.
  - Having dependencies on apps from a different project behaves similarly to a dependency to another repository.
- All apps in a project will be built in one build pipeline (apps sorted after dependencies).
- All projects will be built simultaneously (if GitHub runners are available).
- Projects should not have dependencies on each other:
  - Currently, you cannot control the order in which projects are built.
  - You can request apps from other projects to be built with the current project.

In the next section, we'll start exploring AL-Go for GitHub and see how to create a new AL extension, starting from the AL-Go for GitHub templates.

# Creating a new per-tenant extension with AL-Go for GitHub

To create a new per-tenant extension with AL-Go for GitHub, navigate to <https://github.com/microsoft/AL-Go-PTE>, select **Use this template**, and then **Create a new repository**:

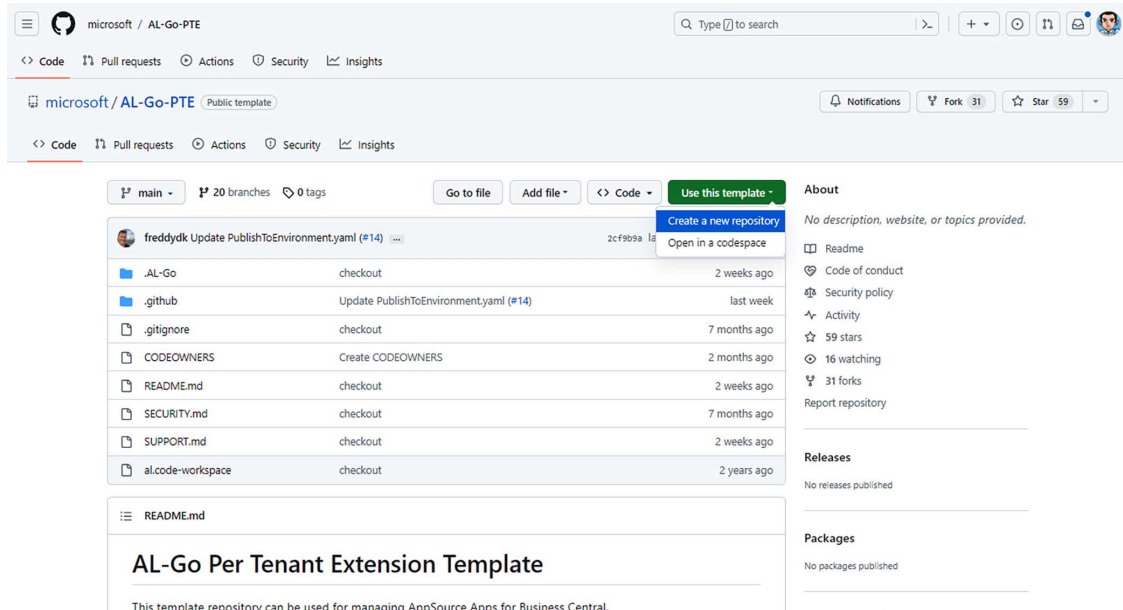


Figure 15.1: Creating a repository from a template

On the **Create a new repository** page, select the name of your extension, set the repository visibility to private or public, and then click on **Create repository**:

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

*Required fields are marked with an asterisk (\*).*

### Repository template

 microsoft/AL-Go-PTE ▾

Start your repository with a template repository's contents.

☐ **Include all branches**

Copy all branches from microsoft/AL-Go-PTE and not just the default branch.

Owner \*



 demiliani ▾

Repository name \*

/ D365BCPTEDemd


Great repository names are short and memorable. Need inspiration? How about [legendary-octo-dollop](#) ?

Description (optional)

- ☐  **Public**  
Anyone on the internet can see this repository. You choose who can commit.
- ☒  **Private**  
You choose who can see and commit to this repository.

### Grant your Marketplace apps access to this repository

You are subscribed to 1 Marketplace app.

☒  **Azure Pipelines** (auto-installed)  
Continuously build, test, and deploy to any platform and cloud

 You are creating a private repository in your personal account.

**Create repository**

Figure 15.2: Configuring the new repository

A new repository will be created in your GitHub account, starting from the Microsoft template. The new repository has a `.AL-Go` folder and other files in it, but no `.al` files:

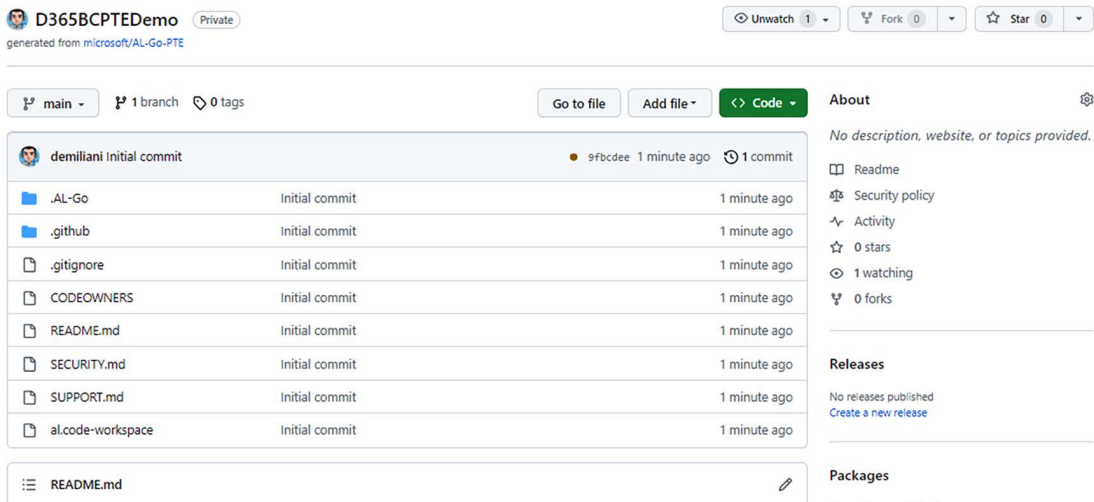


Figure 15.3: New repository with `.AL-Go` folder

From the newly created repository, select **Actions**, and then execute the **Create a new app** action. Here, enter the **Name**, **Publisher**, and **ID range** details for your app, and specify **Y** in **Direct COMMIT**. Then, choose **Run workflow**:

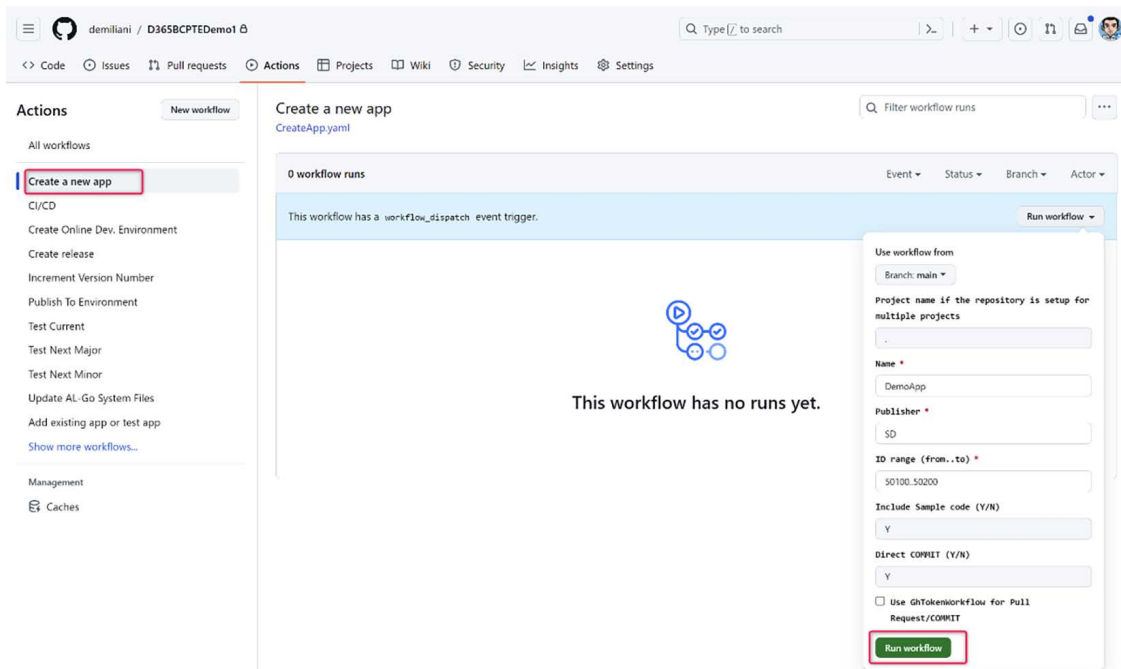



Figure 15.4: Creating a new app



Please note that even though this creates the same files as the regular “AL: Go!” command, this also updates internal pipeline files.

A new AL extension skeleton will be created in your repository. You can monitor the action execution by clicking on the action name:

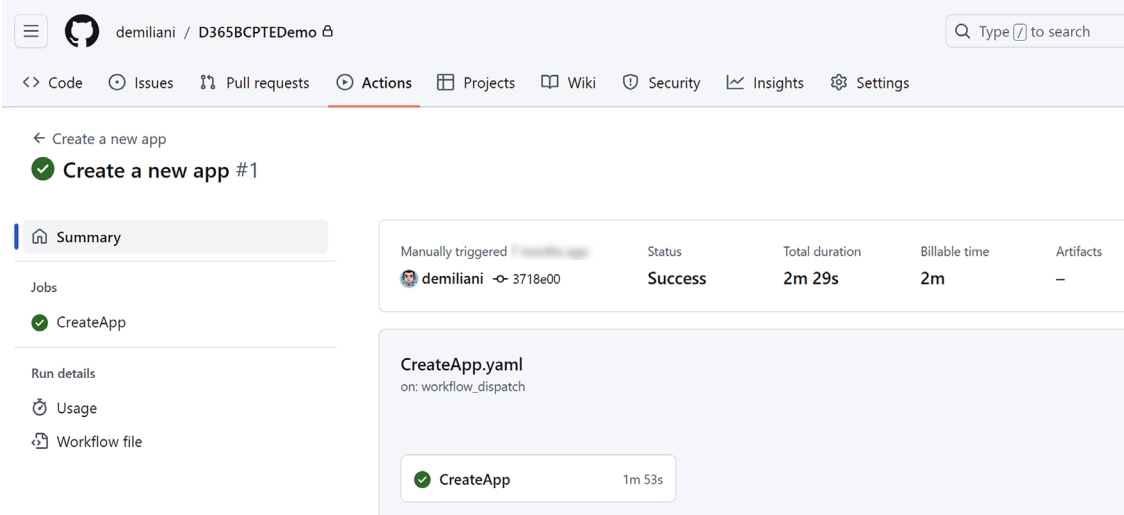


Figure 15.5: Monitoring the CreateApp action

## Branching strategies

Now, you can create branches according to your policy and clone the main branch locally to start working. **Branches** allow you to develop features, fix bugs, or safely experiment with new ideas in a contained area of your repository. You always create a branch from an existing branch. Typically, you might create a new branch from the default branch of your repository. You can then work on this new branch in isolation from changes that other people are making to the repository.

To do that:

1. Start **VS Code**, press *Ctrl+Shift+P*, and select **Git Clone**. Paste the clone URL, and select a folder in which you want to clone the directory.
2. **Open the cloned repository and the workspace** when VS Code asks you (or do it manually)
3. In the `.AL-Go` folder, choose the `localDevEnv.ps1` script, and run the PowerShell script if you want to create a local Docker container for development (this step is not mandatory).

When you start working with code and repositories, some of the initial questions you will ask are, how can you use branches in Git to support your work? Which strategy should I use? When do I create a new branch? When do I merge branches?

There are plenty of strategies you can use, and there is no silver bullet solution. The best strategy for your team could be different for another, or you can have one strategy for your AppSource app and another for your Per-Tenant app.

Before you decide which way you will go, think about the KISS principle (Keep It Simple, Stupid).

In all examples here, we will take master branch as the most stable one, which represents the app as it is released to production. You can decide to name this branch differently, and it has no impact on the strategies themselves. You just need to define the name and make it consistent throughout the team.

## Master-only branch

Having one branch is the simplest strategy you can use. If you have only one developer working on an app, there is no need to create a branch. Even when there are more developers, they could still work on one branch, merging their changes into it each time they conflict. But it is hard to keep the product stable because half-done changes could be pushed into the branch. To stabilize the app, you will need to pause the development. Still, you can change the strategy to something else later, when you see the need for it.

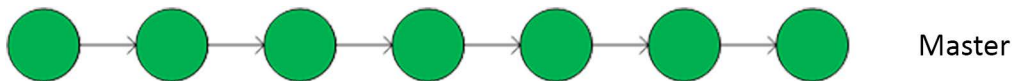


Figure 15.6: Master-only branch diagram

## Feature/developer branches

To isolate changes for one feature or for one developer, you can create a branch for each of them. In this way, the developers work on their own branch, having no conflicts with others until they hit the point when the work is done, and they integrate the changes back into the master branch. After the feature is finished and integrated back into the master branch, the feature branch can be deleted. If the branch is per developer, you can expect the branch to live for a long time. This could be a problem in the long term. Using a branch per developer will mix different changes for different features together, and it could be a problem to release only selected features later in the process. Using branches per feature gives you the possibility to release only selected features or cancel the feature development at no cost (before it is reintegrated).

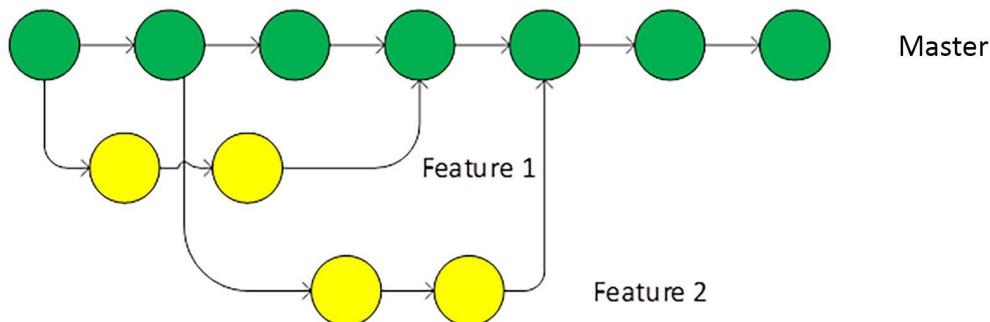


Figure 15.7: Feature branch diagram



This flow is suitable for developing apps for AppSource because the release could be isolated, and you can support multiple versions of the app easily.

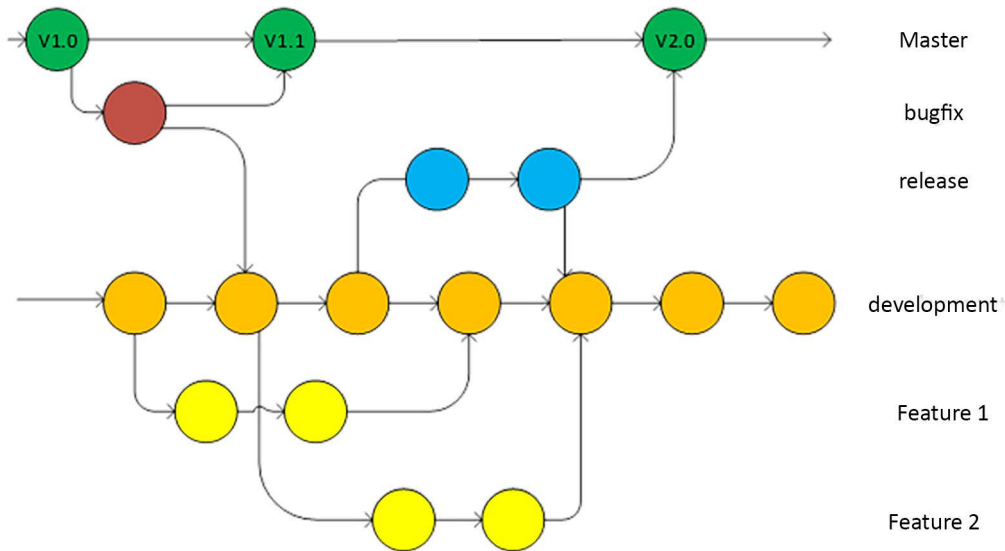


Figure 15.9: Git flow diagram

## GitHub flow

GitHub flow is a workflow based on two main rules:

- Everything in the master branch is releasable anytime
- The release could be done anytime, even multiple times per day

This is basically feature branching. Bug fixing is done like any other development of features. It requires the automatized release of the product.

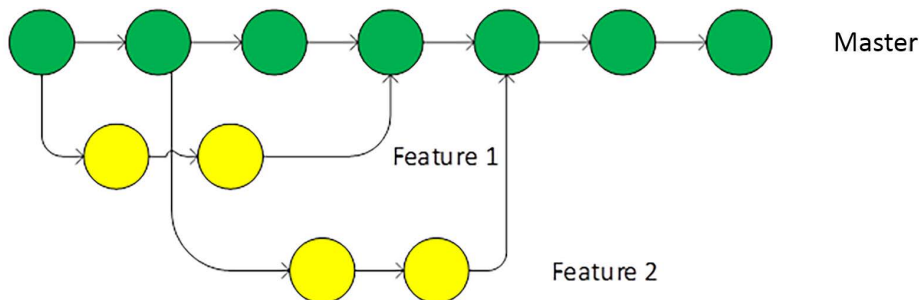


Figure 15.10: GitHub flow diagram

Generally, you can use multiple branching strategies in your company because each is suitable for a different situation. But do not forget KISS! Having a complicated strategy that brings nothing to the team leads to shortcuts and not sticking to the rules. Having one branch is a strategy too. Start with it and add other branches as needed. The strategy could grow with your team.

## Git merge strategies

We will not go deep into all the possibilities of the `git merge` command; rather, we will explain a few terms used in connection with Git and merging.

### Fast-forward merge

When you merge two branches in Git, and one branch is a subset of the commits of the second one, the result will be a **fast-forward (ff)** merge, where no merge is done at all. The branch will just reset to its new position.

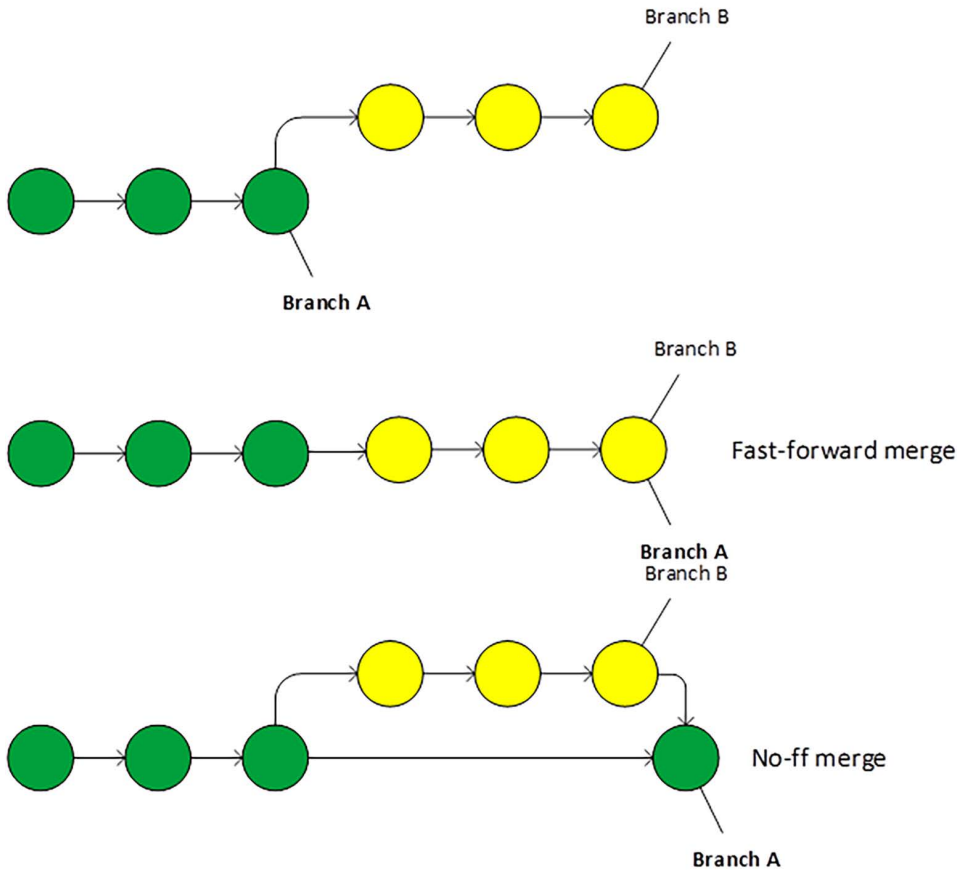


Figure 15.11: Fast-forward merge diagram

### Squash commit

Using squash commit could help you to keep a branch clean and simple. When you want to merge one branch with another, by using squash commit you can join the commits in the branch into new one commit, with a new commit message, and connect this new commit to the target branch. You will lose details but gain simplicity.

It is up to you and what your priority is.

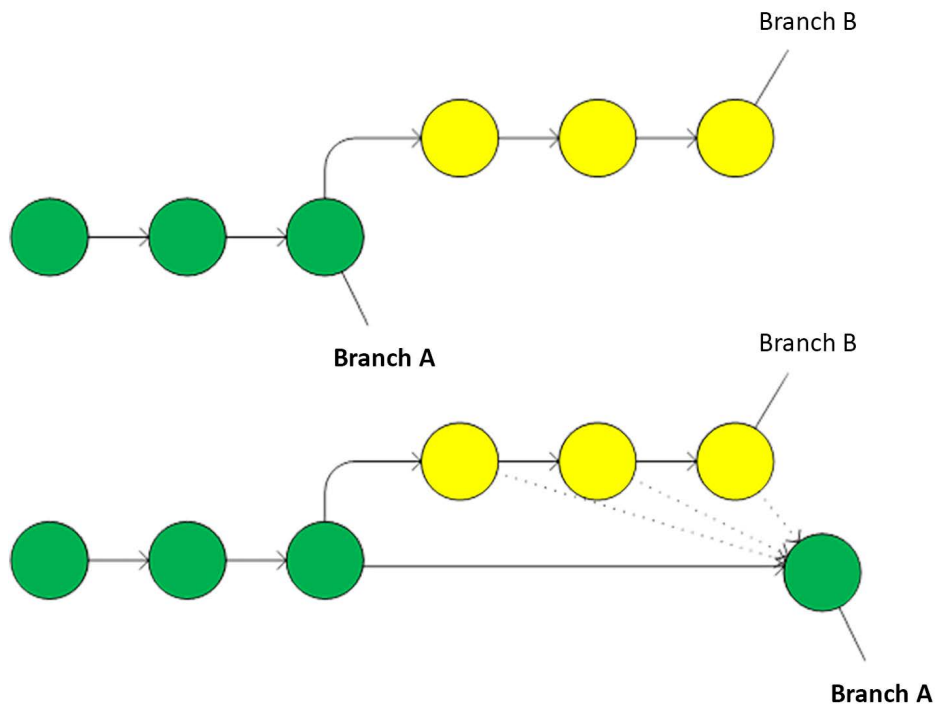


Figure 15.12: Squash commit diagram

## Rebase

Instead of merging, you can use the **rebase** command. As the name suggests, you will take the branch, cut it from the tree, and re-base it on another commit. In this way, you can “base” your changes on the new version without merging. All the commits between the original base and the branch head will be taken and “reapplied” to the new base. With this method you gain simplicity, but you lose details.

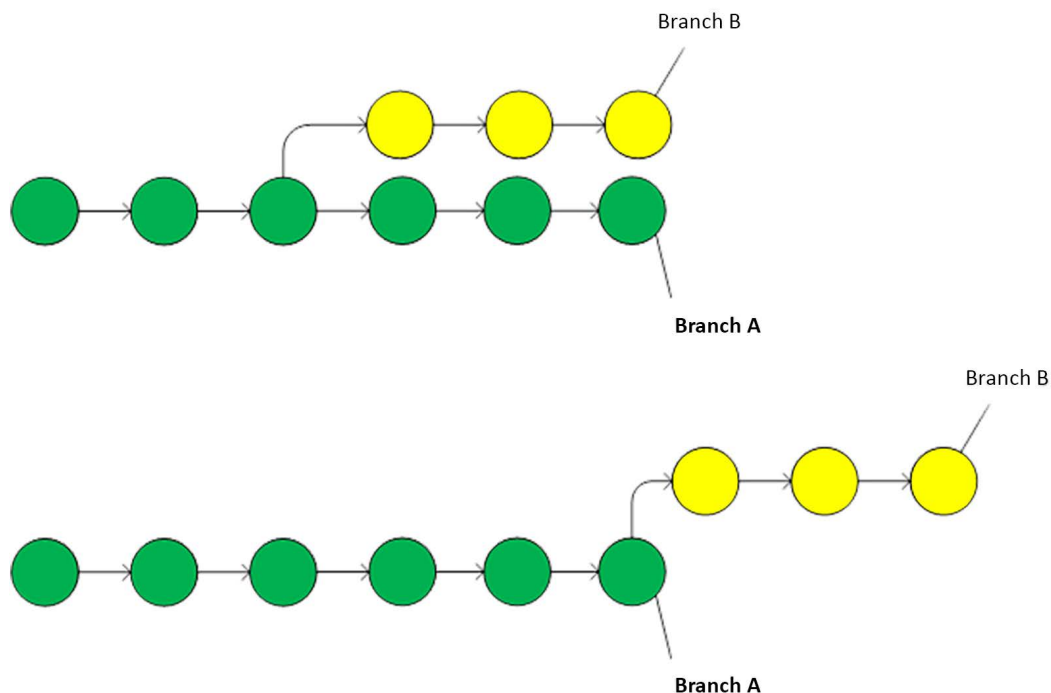


Figure 15.13: Rebase diagram

You can choose if you will use fast-forward merges or not, if you will merge or re-base, or use squash commit to integrate changes back to the target branch. By combining these techniques, you can have a simple history in Git, but you can lose essential details. But again, be aware that you have choices. Just start with the simplest way, and you can change the rules later and see ultimately how they can help you.

## Git in Visual Studio Code

Git is the default **source code management** (SCM) tool for Visual Studio Code, which can begin using from the Visual Studio GUI after installing Git on your system (Git for Windows can be installed from the following link: <https://git-scm.com/download/win>). However, you can enjoy further benefits from installing extensions that will enrich the integration. I recommend using the **GitLens** extension that adds many functions such as different views for Git history, blame functionality, and more.

## Visual Studio Code GUI for Git

First, we will look at the Visual Studio Code interface:

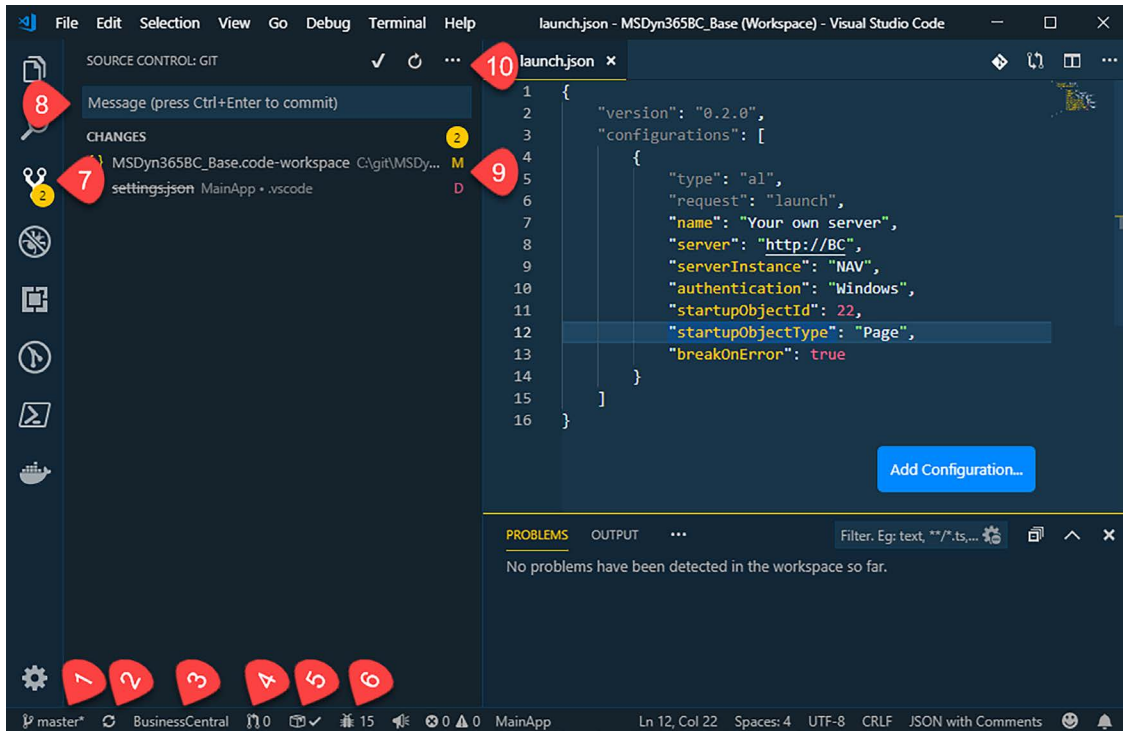


Figure 15.14: Visual Studio Code interface


Let's take a closer look at the points numbered in the above screenshot:

1. The current branch (HEAD) – by clicking on this, you can create new branch or checkout another existing branch (switch to the branch)
2. The status of the repository – we should be in sync with the remote repository. If not, you will see the number of incoming and outgoing commits of the current branch. Clicking this will synchronize the fetch and merge commands with the remote repository.
3. The name of the project in Azure DevOps (the Azure Repo extension) – clicking this will open the Azure DevOps portal.
4. The number of pull requests (the Azure Repo extension) – by clicking this, you can select and browse the pull requests.
5. The last build status (the Azure Repo extension) – by clicking this, you can open the last build for the repository.
6. The number of work items (the Azure Repo extensions) – by clicking this, you can browse the work items and open them in web portal.
7. The source control activity bar – you can see the number of changed files. By clicking this, you will switch the activity to source control, where you can commit the changes.

8. The commit message textbox – enter the commit message before you commit the changes.
9. The list of changes – you can select which changes you want to undo or stage for committing. By double-clicking, you can open the diff window that shows you the differences from the last committed state.
10. The Source Control menu – you will find more commands regarding source control here.

## Workflow

To start using Git in Visual Studio Code, you just need to perform the following steps:

1. The first step is to get the repository on which you want your local system to work:
  - If you want to work on existing code, you need to have URL of the remote repository. Then, you can use the command `Git: Clone`, enter the URL, and select the folder where the repository will be cloned (it will be cloned into the subfolder with the name of the repository).
  - If you are creating a new app, you can first create the folder, open it in VS Code, create the basic structure (e.g., using `AL: GO!` or another command), and then use `Git: Initialize Repository` to make the folder in the Git repository. Later, you can connect the local repository to a new remote one by using Command Prompt.
2. Check out the existing branch or create a new branch on which you want to do your development. You can do this by clicking the branch button at the bottom of the screen. Each time, do not forget to check that you are working on the correct branch.
3. After you make some changes, go to the **Source Control Activity Bar** (press `Ctrl+Shift+G`), write a meaningful message like `My first commit`, and commit the changes by ticking the mark over the message or by pressing `Ctrl+Enter`. If you didn't stage some changes (selected a changed file in the changes list and moved it into **Staged Changes**), VS Code will ask you if you want to commit all changes instead. I recommend going through the changes, manually checking them, and staging them. By staging changes, you can select only a subset of all the modifications that will be committed. You can even stage and unstage online if you open the diff window, right-click on the lines, and select **Stage/Unstage Selected Ranges**. In this way, you can split your changes into separate commits. This is helpful if, for example, they are related to different requirements.
4. If you want to undo the changes, just click on **Discard Changes** in the changed file line.
5. After you commit the changes and you want to make them available for others, you can click on the **Synchronize** button at the bottom of the window (, which will push the commit to the remote repository (and pull changes from it if there are any). You can also set up Visual Studio Code to automatically push new commits to the remote.
6. If you want to merge your changes into the development branch (or any other branch you are not responsible for), create pull requests. You can do this through the **Browse your pull request** option when you click on the **Pull Request** button in the status bar (number 4 in the previous screenshot).

If you need to fix something, for example, because there is a conflict during the pull request, just do the change, commit, push, and the pull request will be updated with the new commit automatically.

7. If everything is saved in the remote repository and the change is merged, you can just delete the folder on your disk if you do not need it anymore.



It is good to make rules on what the commit message should look like to be consistent organization-wide. Mastering the skill to write good commit messages should be part of every developer's self-improvement. You can find some examples and rules on the internet on how to write good git commit messages. An example could be like this:

Fix error "Value is incorrect" in Sales posting

Error text was changed to give more context to user and in some cases, solved by finding correct value automatically.

Fix bug #1234

Related to #1258

## Handling the CI/CD pipeline

When you create a repository with *AL-Go for GitHub*, it automatically adds a CI/CD pipeline to it. I suggest protecting the main branch in the GitHub repository by always requiring a pull request before accepting code changes (and also setting the number of approvals for pull requests).

To do that, select your main branch, and under **Protect matching branches** set the **Require approvals** option accordingly:

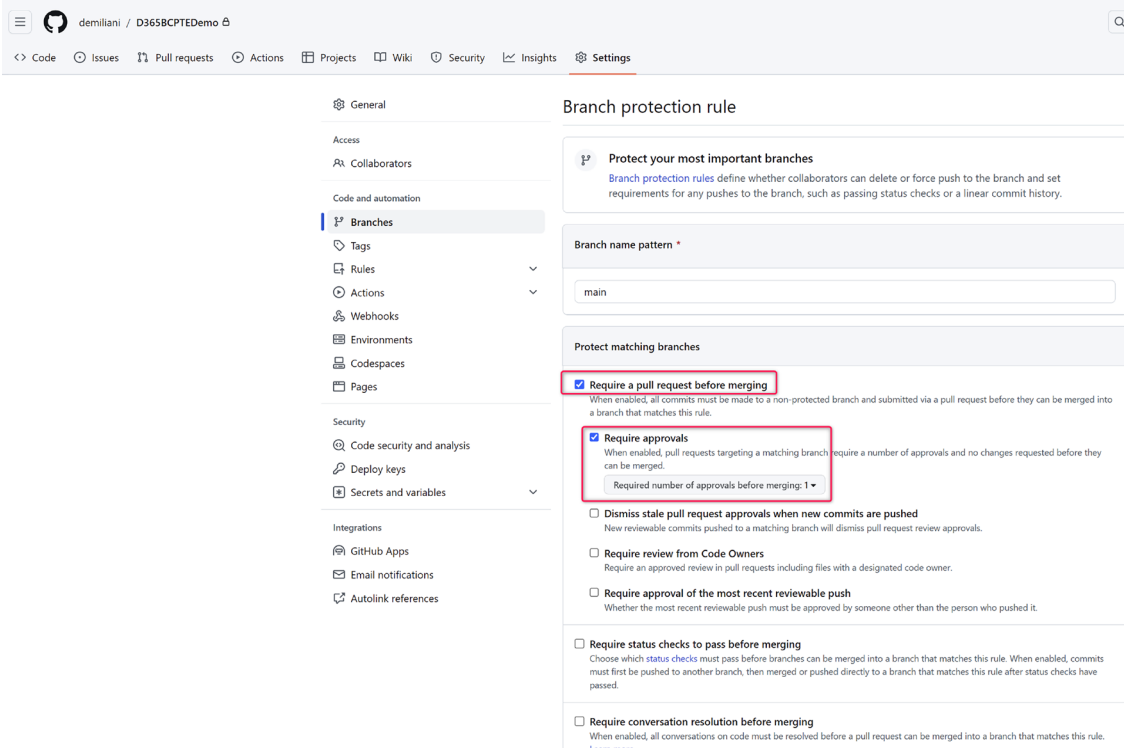


Figure 15.15: Branch protection options

When using pull requests, developers will create a fork or a branch in which they will complete their work. When creating the pull request, their change will be merged with the main branch, and a CI/CD workflow will be run on the merged code before it is pushed into the repository.

If you commit a code modification via a pull request to the *main* branch, you will see that the CI/CD workflow starts:

The screenshot shows the GitHub Actions interface for a repository named 'demiliani / D365BCPTEDemo'. The 'Actions' tab is selected, showing a workflow named 'Pull Request Handler' triggered by a pull request. The workflow status is 'Success'.

**Summary**

Jobs:

- PullRequestHandler
- Build
- CI/CD Workflow

Run details:

- Usage
- Workflow file

**Jobs**

Triggered via pull request	Status	Total duration	Billable time	Artifacts
demiliani opened #1	Success	1m 26s	1m	1

**PullRequestHandler.yml**  
on: pull\_request\_target

**Jobs**

- PullRequestHandler (44s)

**Artifacts**  
Produced during runtime

Name	Size
Pull_Request_Files	400 Bytes

Figure 15.16: CI/CD workflow in operation

This workflow performs the following steps:

1. It creates a Dynamics 365 Business Central Docker container on the GitHub runner (the machine that executes the jobs in a GitHub workflow)
2. Then, it builds and publishes the application
3. If one is present, it runs the test application
4. Finally, it creates the artifact .app file (the output of the build process)

When the CI/CD workflow finishes, you can investigate the steps and the output:

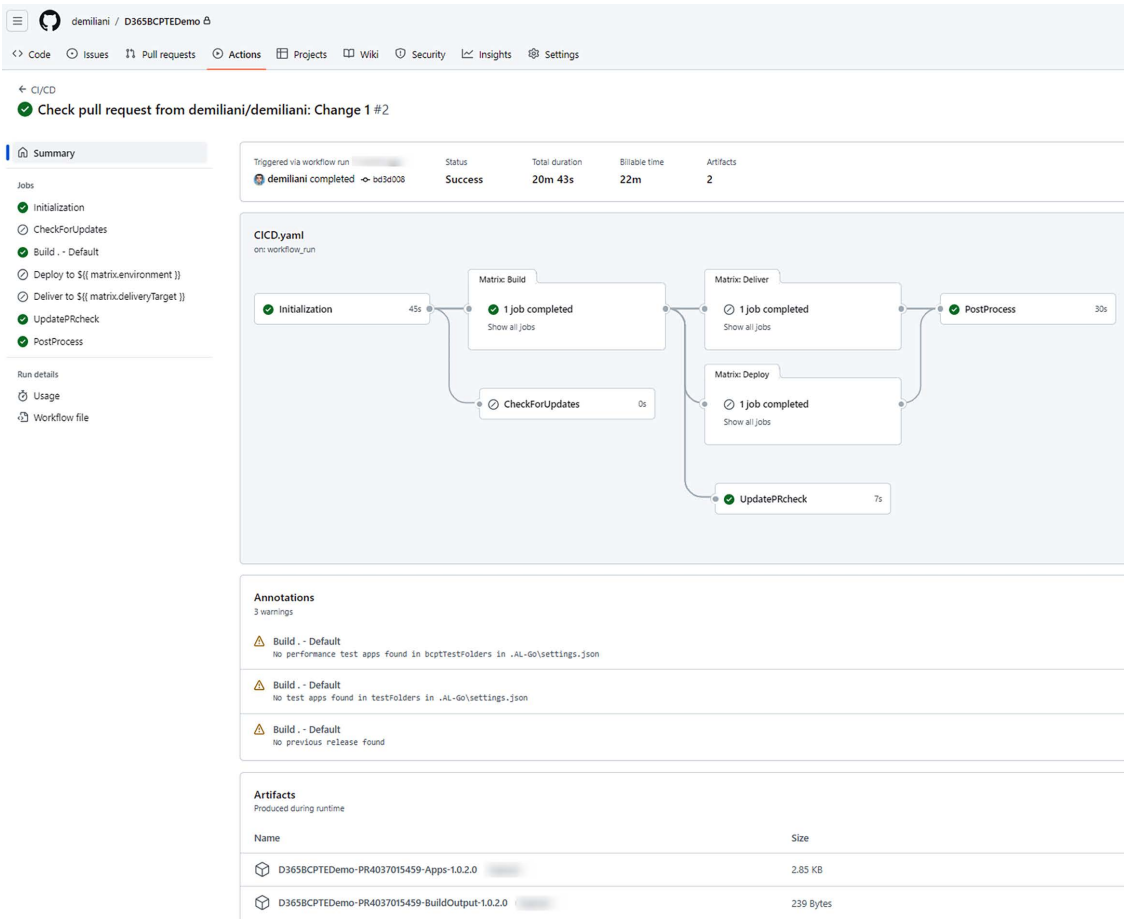


Figure 15.17: CI/CD workflow output

# Setting up your self-hosted GitHub runner

When you execute the CI/CD workflow (the build pipeline), you can see that it takes quite a lot of time, and this is a billable time for GitHub (it consumes the amount of build time you have for pipelines in your account). This is because the pipeline is executed by a GitHub-hosted runner.

You can increase the performance of your build pipelines by adding your own self-hosted GitHub runner, which can cache the generic image, the build image, and also the artifacts.

With a GitHub-hosted runner, you have:

- No caching
- A long build time
- Zero maintenance
- Unlimited minutes of build for public repositories
- Limited minutes for private repositories

With a self-hosted runner, you have:

- Caching of artifacts and images
- A reduced build time
- Maintenance/renewal requirements
- Unlimited minutes of build for public repositories
- Unlimited minutes of build for private repositories

GitHub self-hosted runners can be registered for an organization (accessible for all repositories in the organization) or for a single repository. They can run on your own hardware or on an Azure Virtual Machine.

To register a self-hosted runner, navigate to <https://github.com/organizations/{organization}/settings/actions/runners/new> to create a self-hosted runner for your organization, or use <https://github.com/{organization}/{repository}/settings/actions/runners> to create a self-hosted runner for a single repository. Then, click on **New self-hosted runner**:

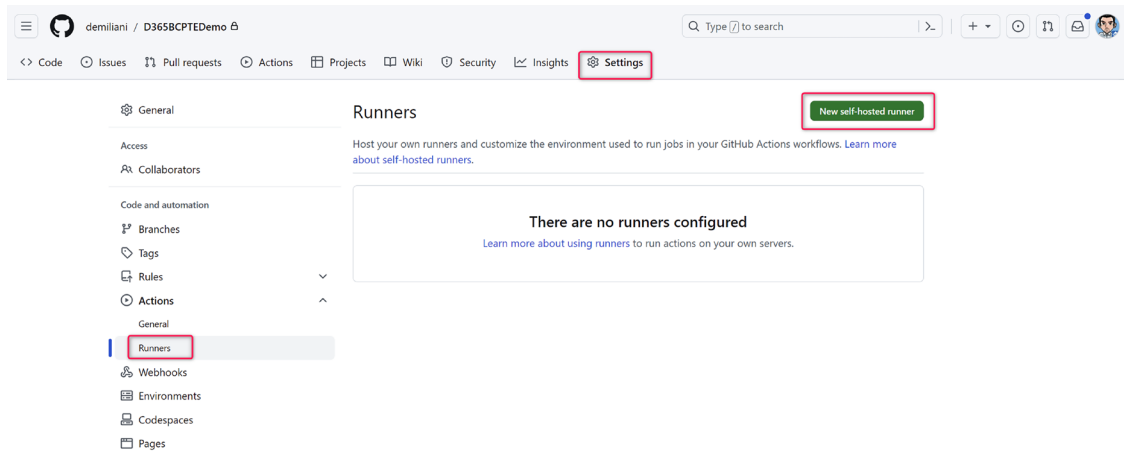


Figure 15.18: Registering a runner for an organization

To create a self-hosted runner manually, choose **Windows** under **Runner Image** and **x64** as **Architecture**, and follow the description on how to create a self-hosted runner manually:

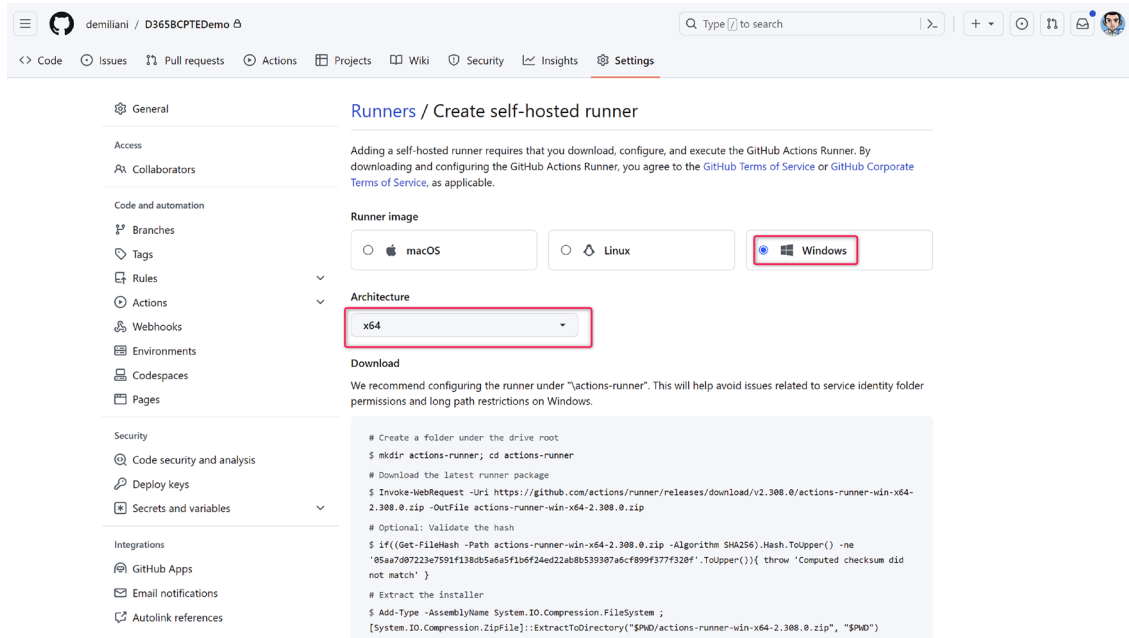


Figure 15.19: Manually creating a runner

You can also create a self-hosted runner by using an Azure Virtual Machine (that can host different runners). To do that, just open the following link:

<https://aka.ms/getbuildagent>

Here, you need to perform the following steps:

1. Enter the **Resource group** name, **Region**, **VM Name**, and **Admin Password** of your choice.
2. Enter the number of agents you want to create on the virtual machine in the **Count** field.
3. Grab the token, the organization URL, and the agent URL from the **Create Self-Hosted runner** page, and specify self-hosted in the **Labels** option.
4. Click **Review + Create**, and then review the deployment and choose **Create**.

- Wait for the Azure VM creation to finalize, and then navigate back to see that the runners have been registered and are ready to use.

[Home](#) >

## Custom deployment ...

Deploy from a custom template

New! Deployment Stacks let you manage the lifecycle of your deployments. Try it now →

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ  ▼

Resource group \* ⓘ  ▼  
[Create new](#)

### Instance details

Region \* ⓘ  ▼

Vm Name ⓘ  ✓

Remote Desktop Access ⓘ  ✓

Operating System ⓘ  ▼

Vm Size ⓘ  ▼

Storage Account Type ⓘ  ▼

Os Disk Size ⓘ  ✓

Vm Admin Username ⓘ  ✓

Admin Password \* ⓘ  ✓

Count ⓘ  ✓

Token \* ⓘ  ✓

Organization \* ⓘ  ✓

Labels Or Pool ⓘ  ✓

Agent Uri ⓘ  ✓

Install Hyper V ⓘ  ✓

Run Inside Docker ⓘ  ✓

Final Setup Script Uri ⓘ  ✓

Figure 15.20: Configuring the runner deployment

Now, on the **Runners** list on GitHub, choose the runner group **Default** and allow public repositories if your repository is public:

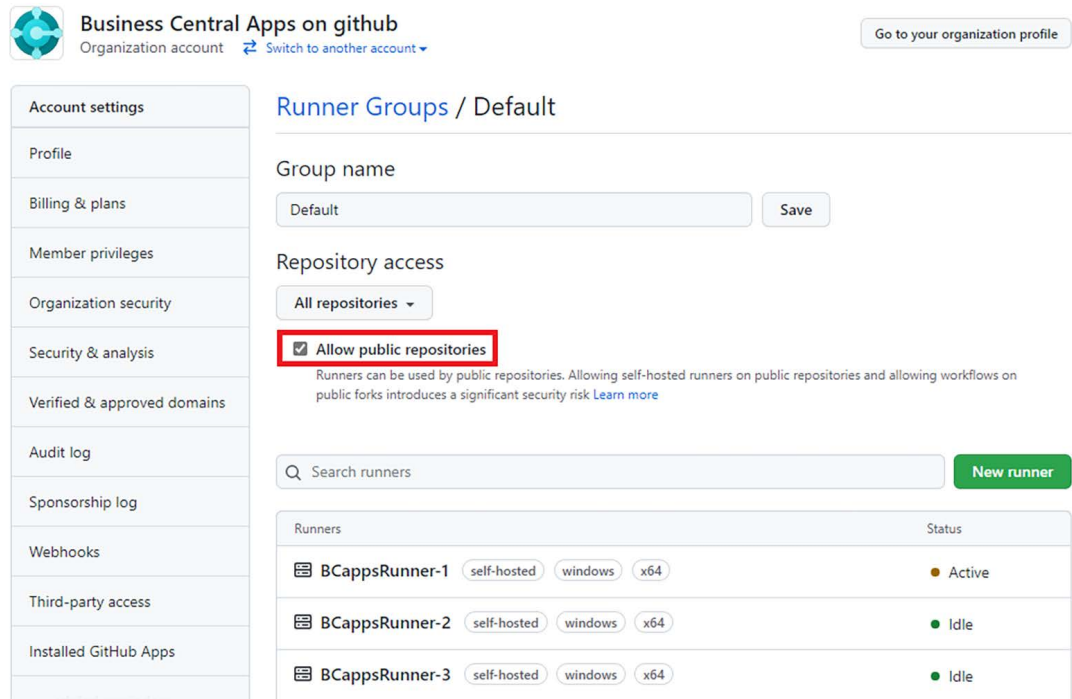


Figure 15.21: Allowing public repositories for a runner

Then, navigate to your project settings file (`.AL-Go/settings.json`) and add the parameter `githubRunner` with the value set to `self-hosted`. Then, save it. Jobs will now be executed on your self-hosted agent.

## Handling dependencies between applications

In real-world Dynamics 365 Business Central projects, it's absolutely common to create extensions with dependencies from other extensions. When working in teams of developers, a best practice when you have multiple extensions is to create separate repositories; one repository should contain one extension.

For **AL-Go for GitHub** to handle the CI/CD pipeline of an extension that has a dependency from an extension stored in another repository, the dependent repository must be added to the dependency probing paths (`appDependencyProbingPaths` parameter) in the **AL-Go** settings file.

To show that, in my GitHub repository, I've created a new per-tenant extension (named `DependendApp`) with the **AL-Go for GitHub** template, as previously explained.

In the `app.json` file of this second extension, I've added a dependency from another app (the previously created `DemoApp`) that is physically stored in another repository on GitHub:

```
1  {
2    "id": "8cf9bb7d-95f8-472e-9910-ef7fb53bb38b",
3    "name": "DependentApp",
4    "publisher": "SD",
5    "version": "1.0.0.0",
6    "brief": "",
7    "description": "",
8    "privacyStatement": "",
9    "EULA": "",
10   "help": "",
11   "url": "",
12   "logo": "",
13   "dependencies": [
14     {
15       "id": "8a2afe1b-a585-4617-88aa-c500266e470f",
16       "name": "DemoApp",
17       "publisher": "SD",
18       "version": "1.0.0.0"
19     }
20   ],
21   "screenshots": [],
22   "platform": "1.0.0.0",
23   "application": "19.0.0.0",
24   "idRanges": [
25     {
26       "from": 50100,
27       "to": 50200
28     }
29   ],
30   "resourceExposurePolicy": {
31     "allowDebugging": true,
32     "allowDownloadingSource": false,
33     "includeSourceInSymbolFile": false
34   }
35 }
```

Figure 15.22: Adding an app dependency

To handle the dependency for the CI/CD pipeline, go to the `.AL-Go` folder and edit the `settings.json` file by adding the following part (referencing the repo where the dependency app is stored):

```
1  {
2    "country": "us",
3    "appFolders": [],
4    "testFolders": [],
5    "bcptTestFolders": [],
6    "appDependencyProbingPaths": [
7      {
8        "repo": "https://github.com/demiliani/D365BCPTEDemo",
9        "version": "latest",
10       "release_status": "latestBuild",
11       "authTokenSecret": "GHTOKENWORKFLOW",
12       "projects": "*"
13     }
14   ]
15 }
```

Figure 15.23: Adding a dependency to the dependency probing paths

Here, we can see that the probing path constitutes the following details:

- `repo` specifies the URL of the repository where the extension added as the dependency is stored.
- `version` specifies the version of the dependency to be downloaded. It could be set to `latest` or a specific version.
- `release_status` specifies the type of release on the foreign repository. The artifacts can be downloaded from a release, prerelease, or a draft.
- `authTokenSecret`: If the foreign repository is private, to download the artifacts, an access token is needed. In this case, a secret should be added to GitHub secrets or Azure Key Vault, and the name of the secret should be provided in the settings.
- `projects` specifies the project in a multi-project repository. `*` means all projects.

Please note that AL-Go doesn't support anybody changing the AL-Go system files and will warn about such changes. To update the AL-Go system files using the **Update AL-Go System Files** workflow, you need to provide a secret called `GHTOKENWORKFLOW`, containing a *Personal Access Token* with permissions to modify workflows.

To do that, navigate to **New personal access token** (<https://github.com/settings/tokens/new>) and create a **New personal access token**.

Name it, set the expiration date, and check the **workflow** option in the list of **scopes**:

**GitHub Apps**  
**OAuth Apps**  
**Personal access tokens** Beta  
 Fine-grained tokens  
 Tokens (classic)

### New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

**Note**

Github Workflow Token

What's this token for?

**Expiration \***

30 days ▼ The token will expire on Fri, Sep 29 2023

**Select scopes**

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> <b>repo</b>	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input checked="" type="checkbox"/> <b>workflow</b>	Update GitHub Action workflows
<input type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry
<input type="checkbox"/> delete:packages	Delete packages from GitHub Package Registry
<input type="checkbox"/> admin:org	Full control of orgs and teams, read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership, read and write org projects
<input type="checkbox"/> read:org	Read org and team membership, read org projects
<input type="checkbox"/> manage_runners:org	Manage org runners and runner groups
<input type="checkbox"/> admin:public_key	Full control of user public keys

Figure 15.24: Creating a personal access token

Generate the token and copy it to the clipboard. You won't be able to see the token again.

Now perform the following steps:

1. Open **Settings** in your repository and select **Secrets and variables | Actions**.

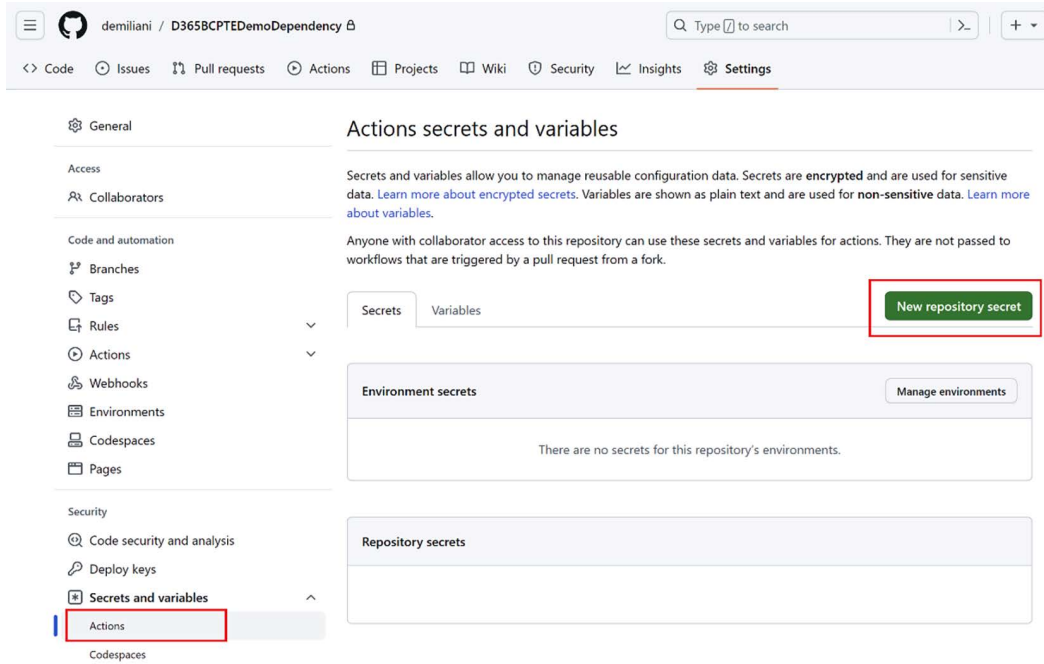


Figure 15.25: Repository secret settings

2. Click on the **New repository secret** button, and create a secret called **GHTOKENWORKFLOW**.
3. Paste the personal access token in the value field, and then click on **Add secret**.

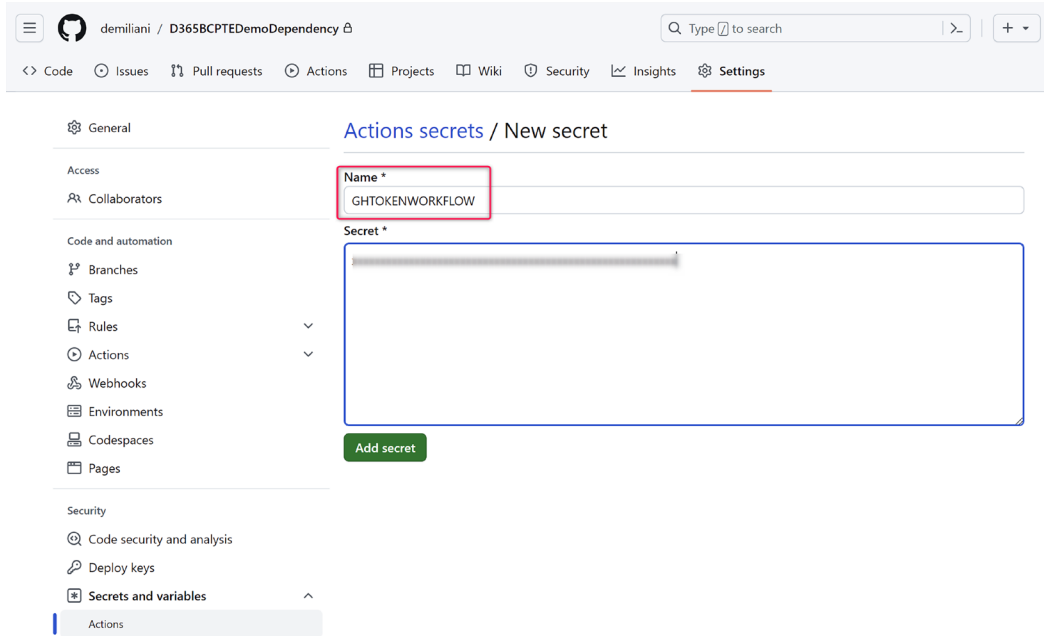


Figure 15.26: Adding the personal access token as a secret

Now, when you commit a modification to the **main** branch (via a pull request), the CI/CD GitHub workflow starts. The required dependent app will be downloaded and built from its repository, all apps will be installed on the build container, and the build pipeline will be executed:

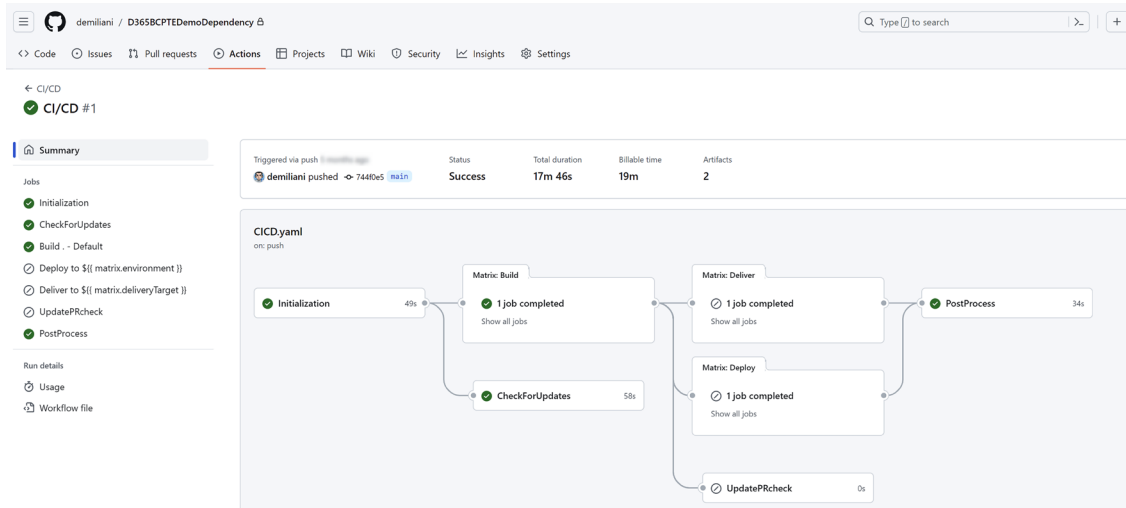


Figure 15.27: Executing the build pipeline

## Adding a test application to an existing project

Creating test applications for automatic testing is important when developing AL extensions.

AL-Go for GitHub permits you to create a test application in the same repository of your application by going to Actions, and then by executing the **Create a new test app** workflow:

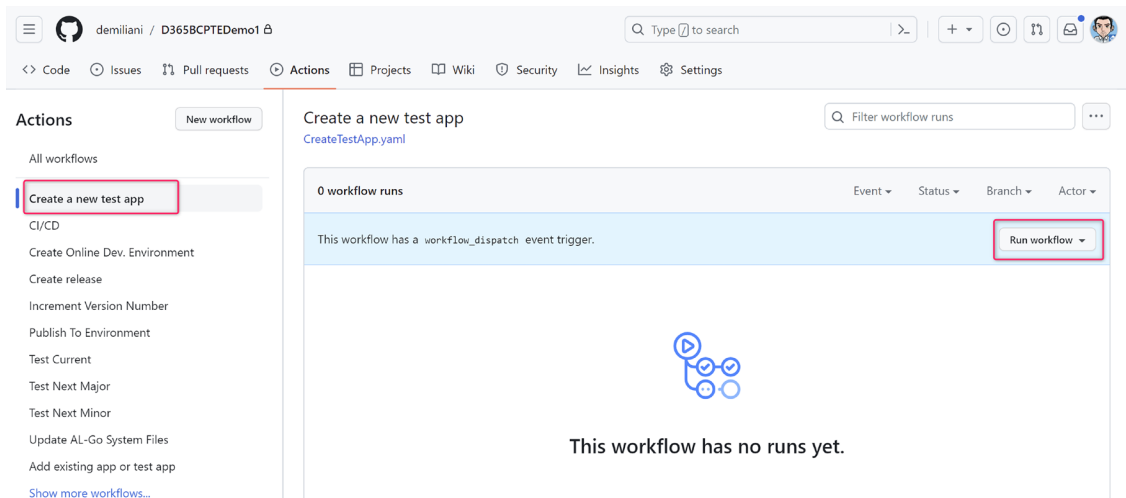



Figure 15.28: Button for creating a test app

Fill in the required parameters, and at the end of the workflow execution, a test application will be added to the repository:

D365BCPTEDemo1 / .D365BCPTEDemo.Test / 

 demiliani 'New Test App (.D365BCPTEDemo.Test)'





Name	Last commit message
 ..	
 .vscode	'New Test App (.D365BCPTEDemo.Test)'
 HelloWorld.Test.al	'New Test App (.D365BCPTEDemo.Test)'
 app.json	'New Test App (.D365BCPTEDemo.Test)'

Figure 15.29: Test application shown in the repository

This will be the starting point to create your own tests.

If a test application is present on the repository, when you trigger the CI/CD workflow, the tests will also be executed.

AL-Go for GitHub does NOT include a visual test results viewer, mainly because GitHub doesn't provide one yet and third-party actions are not considered secure. You will have to download and inspect the test results if they fail or look for them in the workflow output.

## Registering a customer sandbox environment for continuous deployment

In DevOps practice, the **deployment** phase involves publishing and installing an app directly into an environment. Continuous deployment does this on every successful build from the **main** branch.

With AL-Go for GitHub, you can register a customer sandbox environment where you can automatically deploy applications for continuous deployment.



Continuous deployment is only supported in Sandbox environments.

To do this, you need to add a **New environment** in GitHub (environments are supported on Teams or Pro plans only).

To support a continuous deployment process in a customer's sandbox, you first need to configure S2S Authentication to access Automation APIs in the customer's environment. Instructions on how to do that can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/administration/automation-apis-using-s2s-authentication>.

To add a new environment, go into the repository **Settings** and select **Environments**. Then, click on **New environment**:

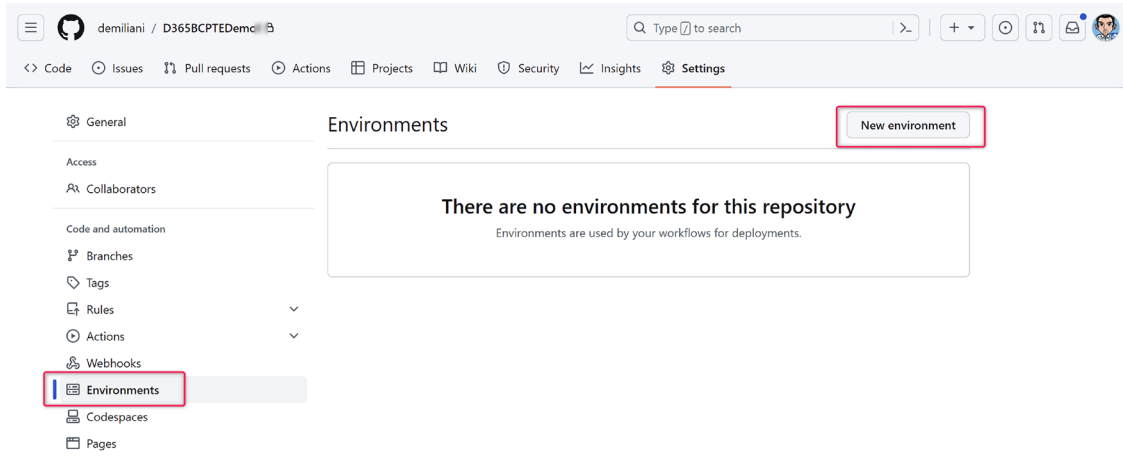


Figure 15.30: Adding a new environment

Provide a name for your new customer's environment, and then click on **Configure environment**:



Figure 15.31: Naming the new environment

Under **Environment secrets**, select the **Add Secret** action. Here, you need to create a secret called AUTHCONTEXT, and enter a compressed JSON string with three values: the Customer's TenantID, the ClientID, and the ClientSecret of the registered AAD application.

The JSON format will be as follows:

```
{ "TenantID": "<TenantID>", "ClientID": "<ClientID>", "ClientSecret": "<ClientSecret>" }
```

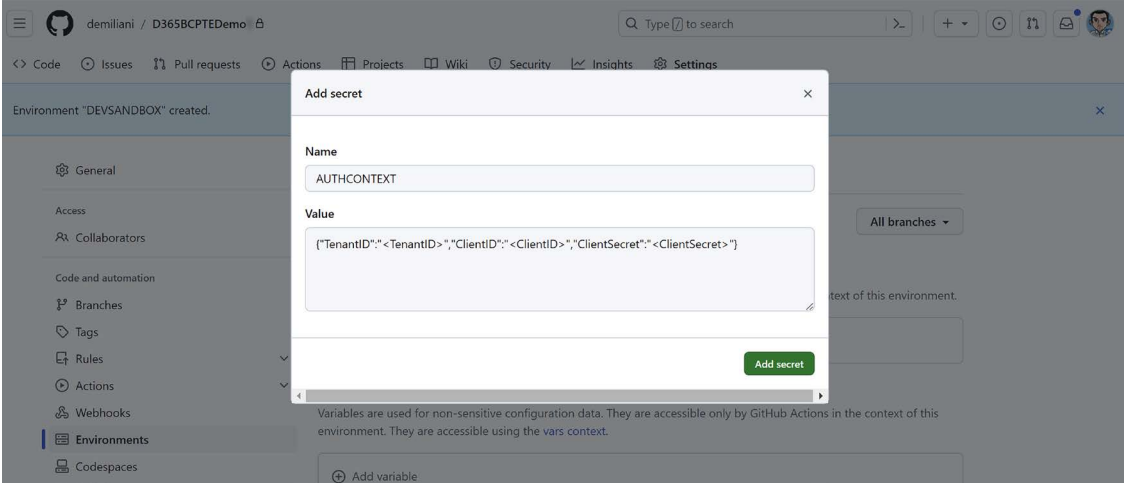


Figure 15.32: Adding a secret to the environment

Click on **Add secret**, and a new environment with an associate secret will be added to the repository:

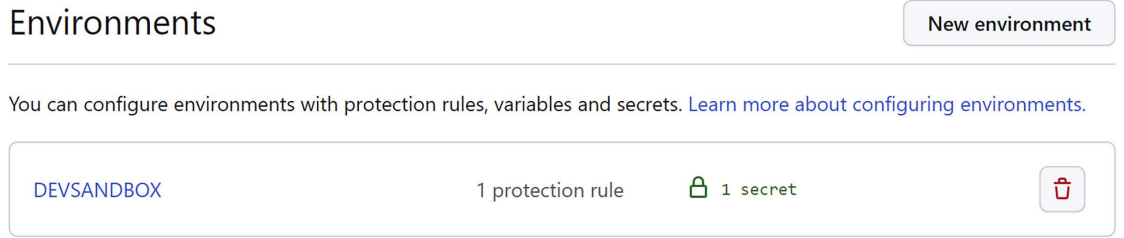


Figure 15.33: Environment with a secret

To start the continuous deployment workflow, navigate to the **Actions** menu, select the **Publish To Environment** workflow, and click **Run workflow**.

Here, select the branch, enter the **latest** (or **current**) in the **App version** field, and insert the name of the environment where you want to deploy the app. Then, click on **Run workflow**:

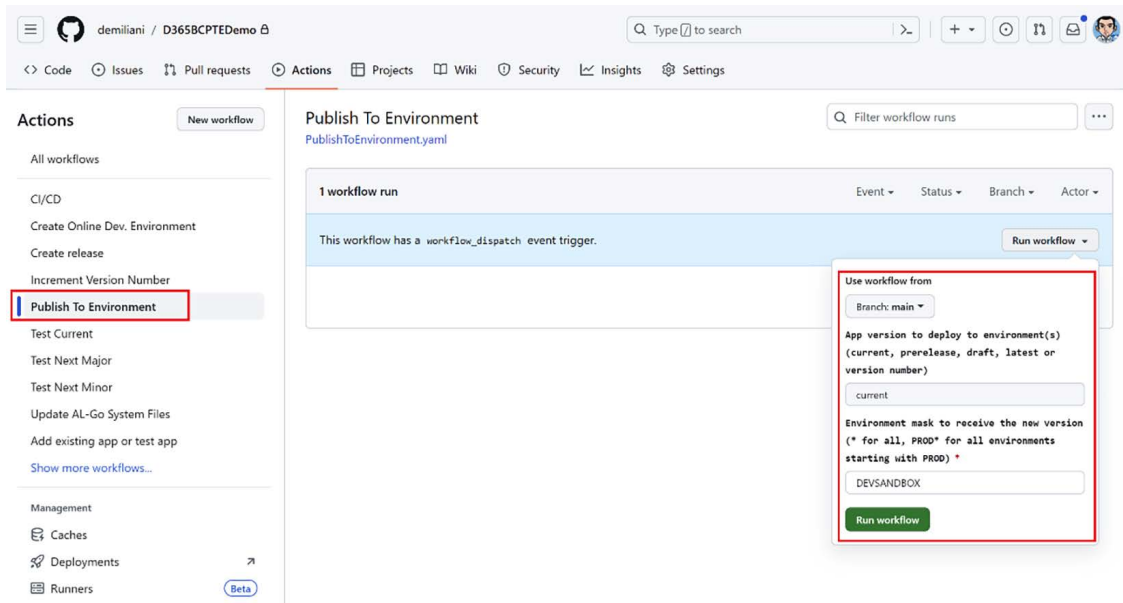


Figure 15.34: Publishing an app to an environment

The **Publish To Environment** workflow will execute, and your application will be published in the customer's sandbox environment automatically:

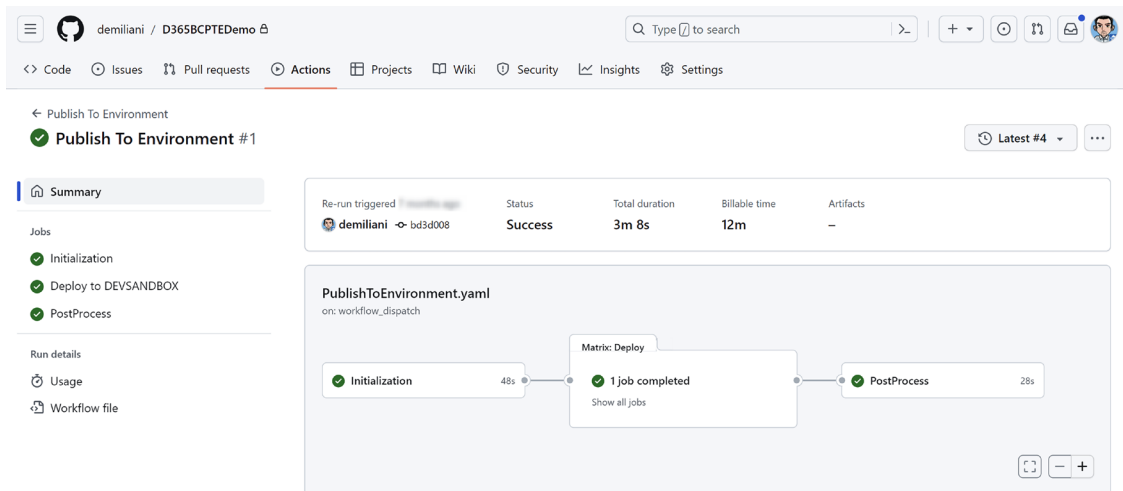


Figure 15.35: Publish To Environment workflow

# Creating a release for your application

A **release** is a deployable software iteration that you can package and make available to others to install or download. Releases are based on Git tags that mark a specific point in your repository’s history.

AL-Go for GitHub supports the creation of releases for your extensions. To create a release, go to the **Action** menu in your repository. Here, select the **Create release** workflow, then click on **Run workflow**, and fill in the required parameters for your release:

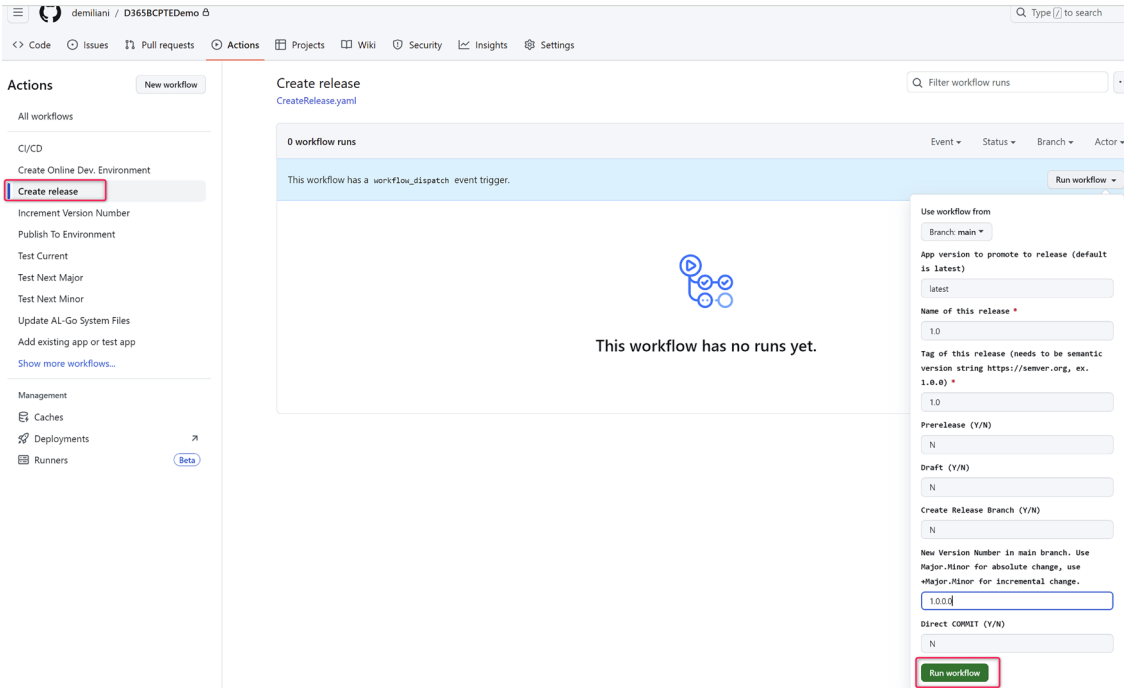


Figure 15.36: Running the Create release workflow

When you click on the green **Run workflow** button, the **Create release** workflow starts. When the workflow is completed, you can go to your repository's **Code** section and see the newly created release:

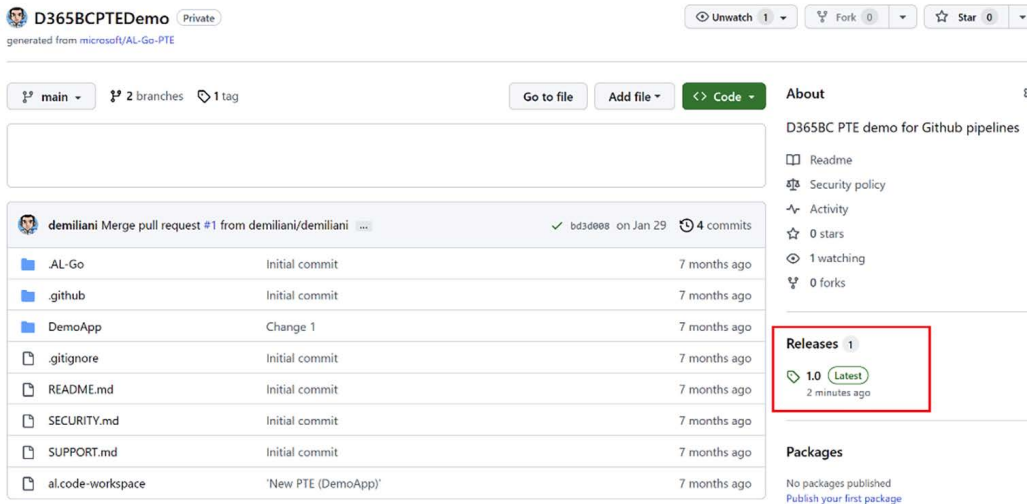


Figure 15.37: Viewing the release label

Choose the newly created release (1.0), and you will see the release details. The release notes are pulled from all pull requests checked in since the last release. The auto-generated release note also contains a list of the new contributors and a link to the full changelog.

At the bottom, you can see the artifacts published: both the apps and the source code. A tag is created in the repository for the release number to keep track of these artifacts.

## Registering a customer Production environment for manual deployment

With AL-Go for GitHub, you can also deploy an extension to a customer's Production environment.

To do that, you first need to add a new environment on GitHub that maps to the customer's Production environment (the procedure is the same as that explained in the *Registering a customer sandbox environment for Continuous Deployment* section to add the environment).

The name of the added environment on GitHub must contain the **(Production)** word:

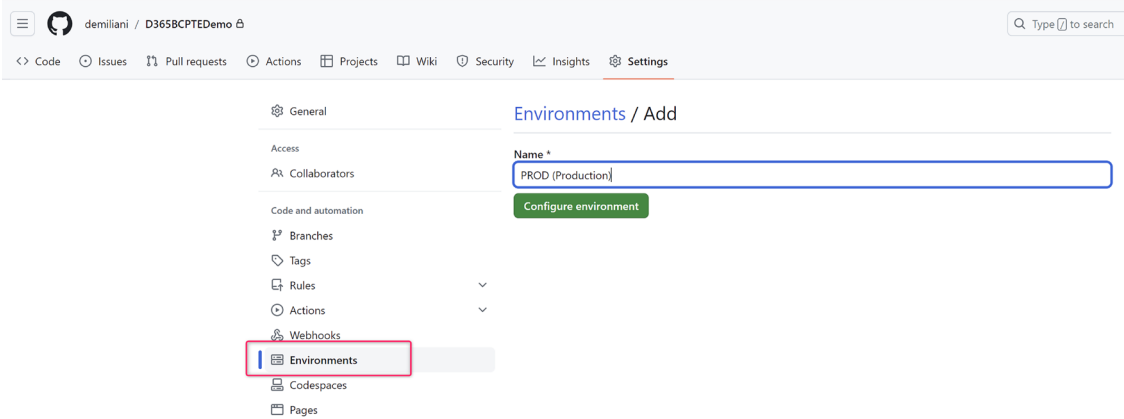


Figure 15.38: Adding a new environment with a specific name

Then, click on **Configure environment**, and set the AUTHCONTEXT parameter as previously explained. By adding the **(Production)** tag, the environment will be filtered out during the **Analyze** phase of the **CI/CD** pipeline. Apps will not be deployed to production environments from the **CI/CD** pipeline; to deploy an app in a Production environment, you need to execute the **Publish To Environment** workflow:

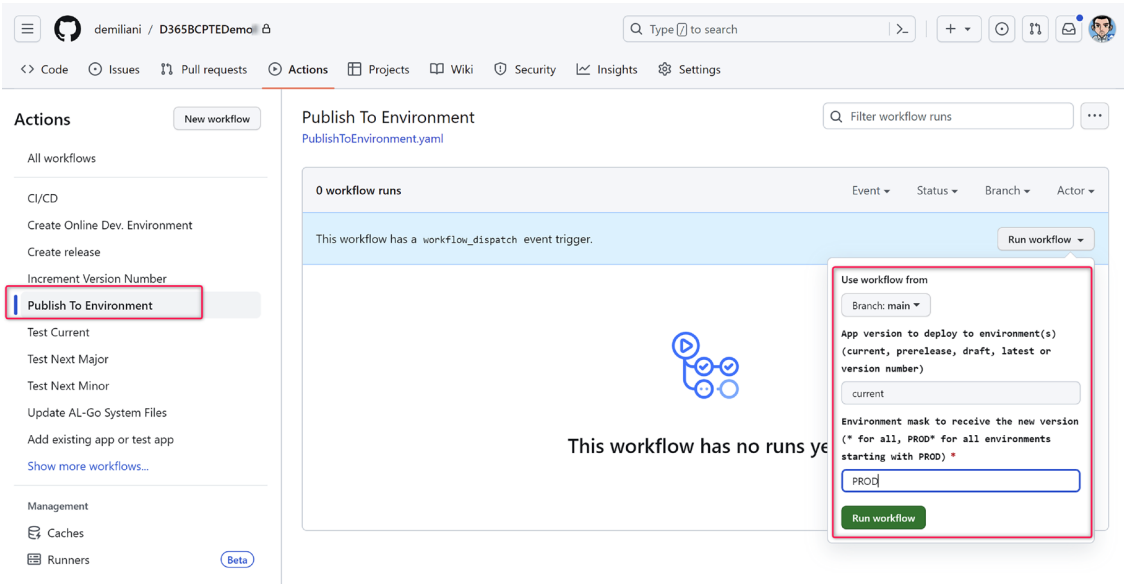


Figure 15.39: Running the Publish To Environment workflow

When the **Publish To Environment** workflow is finished, the app is published in the specified Dynamics 365 Business Central Production environment:

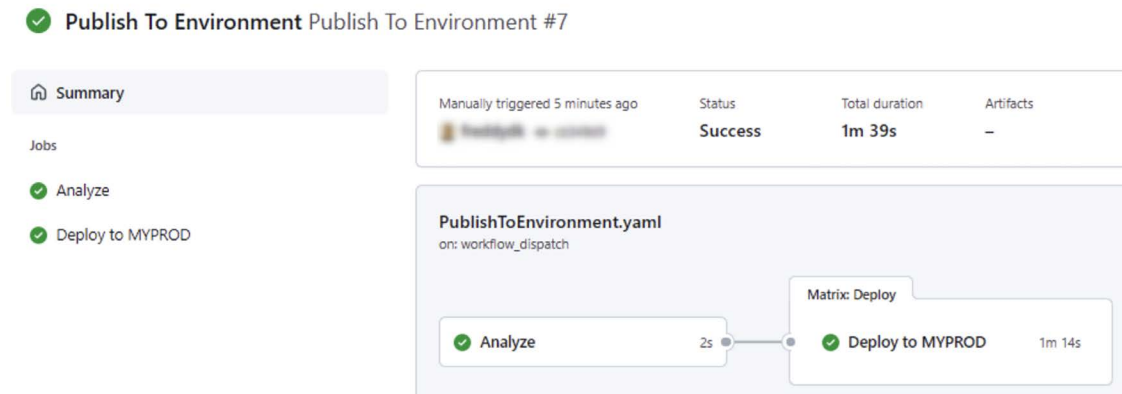


Figure 15.40: Publish To Environment workflow



If your Business Central environment name contains spaces or special characters, you might need to map your GitHub environment name to your Business Central environment name, using the `DeployTo` setting with an `EnvironmentName` setting in the `.AL-Go` file settings. The format is the following:

```

"DeployTo<GitHubEnvironmentName>": {
  "EnvironmentName": "<Business Central Environment Name>"
}
  
```

## Adding a performance test app to your repository

AL-Go for GitHub also supports the creation of a performance test app in an extension's repository. To do that, follow these steps:

1. In your repository under the **Actions** menu, select **Create a new performance test app** and click **Run workflow**. Specify **Name** and **Publisher**, and click the **Run workflow** button again:

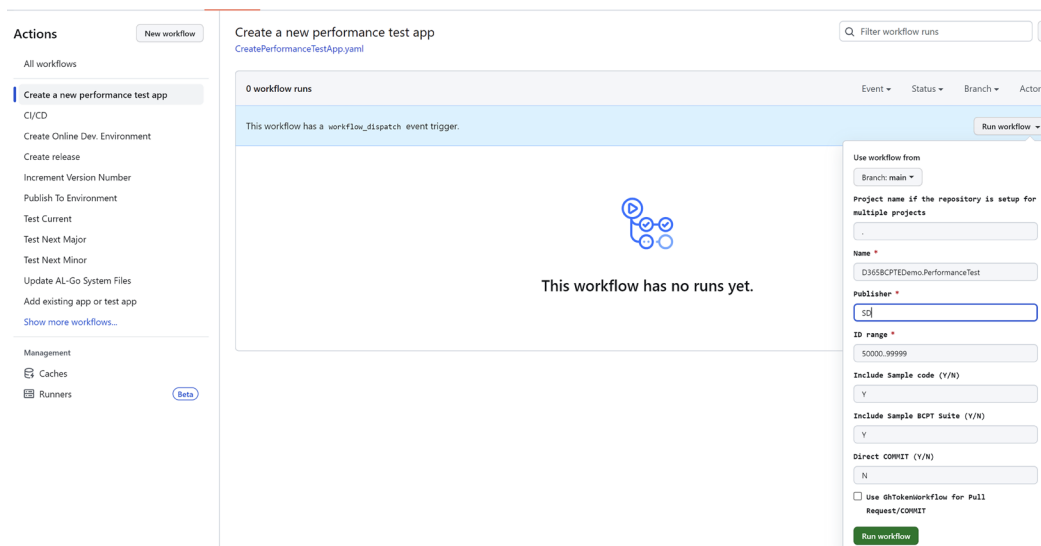


Figure 15.41: Creating a performance test app

2. Inspect the **Pull request** and merge it.
3. When the CI/CD workflow is finished, you will see that performance tests are executed and the results are published as build artifacts. You can control the execution of performance tests by setting the `doNotRunBcptTests` parameter in the `.github\AL-Go-Settings.json` file:

```
"doNotRunBcptTests": false
```

## Using AL-Go for GitHub for AppSource development

AL-Go for GitHub supports creating CI/CD pipelines for AppSource extensions (extensions that targets Microsoft's marketplace).

To create an extension that targets the AppSource marketplace with AL-Go for GitHub, you need to start from the <https://github.com/microsoft/AL-Go-AppSource> repo template and use this as a starting template for your extension. This will create a new extension, exactly like the one previously explained in the *Creating a new per-tenant extension with AL-Go for GitHub* section in this chapter.

To support AppSource code validation (*AppSourceCop* code analysis), you need to go to the `.AL-Go/settings.json` file, and there, add the `AppSourceCopMandatoryAffixes` property with the affix of your AppSource app:

```
"AppSourceCopMandatoryAffixes": [ "PACKT" ]
```

Then, AppSource apps must be digitally signed with a code-signing certificate. To support the code signing, you need to also add to the `settings.json` file two other secrets, pointing to the code-signing .pfx certificate file URL and its corresponding password, such as the following:

```
"CodeSignCertificateUrlSecretName": "YourCodeSignCertUrl",
"CodeSignCertificatePasswordSecretName": "YourCodeSignCertPassword",
```

If you want to use your development license for the CI/CD pipeline, you need also to add a secret containing the secure URL of your license file:

```
"LicenseFileUrlSecretName": "LicenseFile",
```

Please note that the standard Cronus license file now supports building AppSource apps; it has sufficient rights for DevOps, starting from Dynamics 365 Business Central version 22.

You can also use AL-Go for GitHub to automate submissions of apps to the AppSource marketplace. To get started with that, you need to authenticate to the **Microsoft Partner Center Ingestion API**. To do that with S2S, you need to follow step 1 listed at this link: <https://docs.microsoft.com/en-us/azure/marketplace/azure-app-apis>.

At the end of this step, you should have the `ClientID` and `ClientSecret` for your registration.

To set up continuous delivery to AppSource of your extension, you need to create a secret called `AppSourceContext` with the value of the below script:

```
$authcontext = New-BcAuthContext `
    -clientID $publisherAppClientId `
    -clientSecret $publisherAppClientSecret `
    -Scopes "https://api.partner.microsoft.com/.default" `
    -tenantID "YourTenantId"

New-ALGoAppSourceContext -authContext $authcontext | Set-Clipboard
```

Then, you need to create the following settings:

1. `AppSourceProductId`: contains the ID of your extension
2. `AppSourceContinuousDelivery`: set to true

An example of an `.AL-Go/settings.json` file for AppSource continuous delivery is the following:

```
{
  "country": "us",
  "VersioningStrategy": 16,
  "KeyVaultCertificateUrlSecretName": "KeyVaultCertificateFile",
  "KeyVaultCertificatePasswordSecretName": "KeyVaultCertificatePassword",
  "KeyVaultClientIdSecretName": "KeyVaultClientId",
  "CodeSignCertificateUrlSecretName": "CodeSignCertificateUrl",
  "CodeSignCertificatePasswordSecretName": "CodeSignCertificatePassword",
  "AppSourceCopMandatoryAffixes": [
    "PACKT"
  ],
  "AppSourceProductId": "a3f1f921-2ac4-48b2-8ed4-6719c53a0603",
  "AppSourceContinuousDelivery": true,
  "RepoVersion": "4.1"
}
```

The `VersioningStrategy` parameter handles the versioning of your app accordingly using the following schema:

VersioningStrategy	Build	Release
-1	AppJson.Version.Build	AppJson.Version.Release
0	GitHub_Run_Number + Settings.runNumberOffset	GitHub_Run_Attempt - 1
1	GitHub_Run_ID	GitHub_Run_Attempt - 1
2	UtcNow.Format("yyyyMMdd")	UtcNow.Format("HHmmss")
15	Int32.MaxValue	0

Table 15.1: VersioningStrategy schema

If the `VersioningStrategy` is +16, that means that `RepoSettings.RepoVersion` is used as major-minor versioning structure.

The default `VersioningStrategy` is 0.

When all the settings are complete, in your repository **Actions** menu, you will have a workflow called **Publish to AppSource** that you can execute to support automatic delivery to the AppSource marketplace:

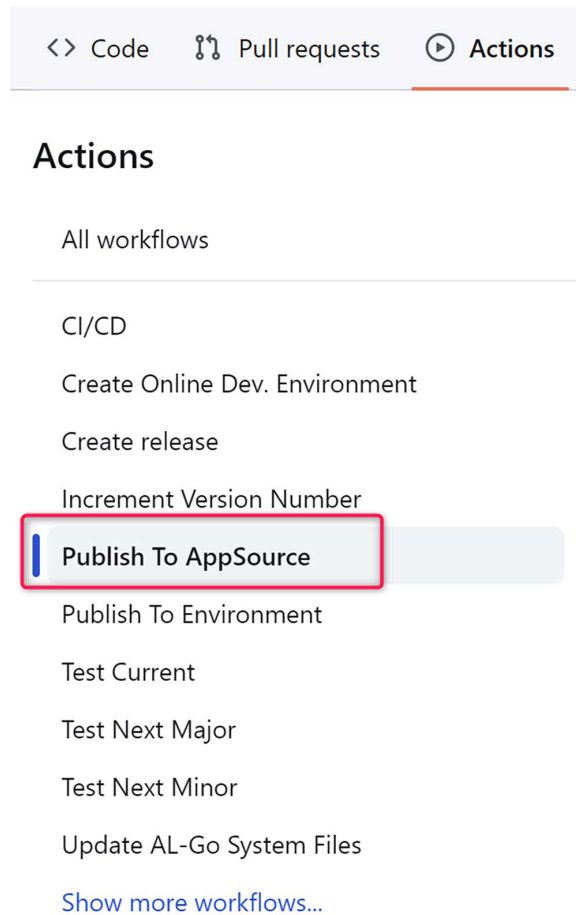


Figure 15.42: Publish to AppSource workflow under Actions

## Summary

In this chapter, we talked about the importance of applying DevOps to Dynamics 365 Business Central extension development, and we provided an overview of the official DevOps tool for Dynamics 365 Business Central offered by Microsoft: AL-Go for GitHub.

We saw how to create extensions with the AL-Go for GitHub templates, how to use repos and branches for SCM, how we can use CI/CD pipelines during extension development, how we can publish applications on a customer's sandbox environment, how to release applications in a production environment, and how to automate the continuous deployment of an extension to the AppSource marketplace.

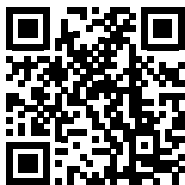
The continuous delivery implementation in AL-Go for GitHub is extensible, and new features will be added in the future. You can also contribute to the project – it's open source – or create your own custom AL-Go for GitHub version.

In the next chapter, we'll see how you can integrate Dynamics 365 with Microsoft Power Platform and how you can create solutions on top of it.

## Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/businesscenter>





# 16

## Dynamics 365 Business Central and Power Platform Integration

Dynamics 365 Business Central is thoroughly integrated with **Microsoft Power Platform**, the big set of low-code tools for building applications, workflows, data analytics, and intelligent bots in Microsoft's ecosystem.

In this chapter, we'll explore the possibility of integrating Dynamics 365 Business Central with Power Platform, with a focus on the following topics:

- Using Power Automate with Dynamics 365 Business Central
- Integrating Power Apps with Dynamics 365 Business Central
- Exposing Dynamics 365 Business Central data to Dataverse by using Dataverse virtual tables
- Exposing Dynamics 365 Business Central events to Dataverse

### Technical requirements

To follow this chapter, you need to have access to the following Power Platform environments:

- For Power Automate: <http://make.powerautomate.com>
- For Power Apps: <http://make.powerapps.com>

Licenses for using these platforms in the context of a Business Central project are included in the standard Dynamics 365 Business Central license.

### Power Automate and Dynamics 365 Business Central

**Microsoft Power Automate** is the Power Platform component for designing user-centric automation workflows. The Dynamics 365 Business Central license gives you the option to use Power Automate with your ERP data for automations and connecting your data to/from internal or external data sources via the Business Central connector.

The **Business Central connector for Power Platform** relies on OData and gives you access to all the Dynamics 365 Business Central entities (standard and custom) exposed as API pages. With this connector, you can create flows that:

- Handle document approvals
- React to Dynamics 365 Business Central entity changes
- Starts with the context of the selected Dynamics 365 Business Central record
- Interact with external applications
- Interact with Dataverse

In this section, we'll see some examples of how to use Power Automate with Dynamics 365 Business Central.

## Example 1: Adding a default image to a Customer record

When a user creates a Customer record in Dynamics 365 Business Central, we want to automatically assign to it a default image (retrieved from a OneDrive folder).

With Power Automate ([make.powerautomate.com](https://make.powerautomate.com)), we can create a new automated cloud flow and then, from the blank template, select the **Dynamics 365 Business Central** connector:

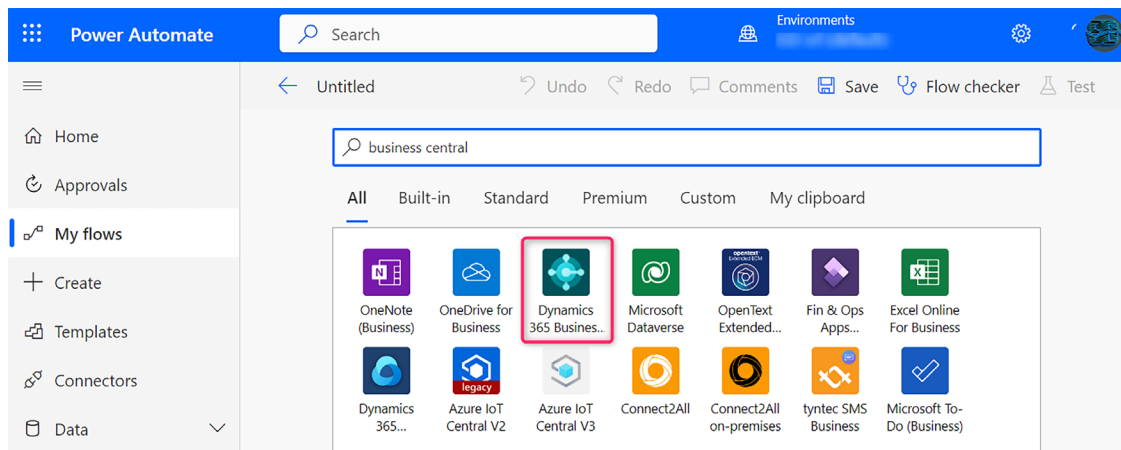


Figure 16.1: Dynamics 365 Business Central connector

When selected, the connector gives you a list of available triggers for your workflow and then we select the **When a record is created (V3)** trigger:

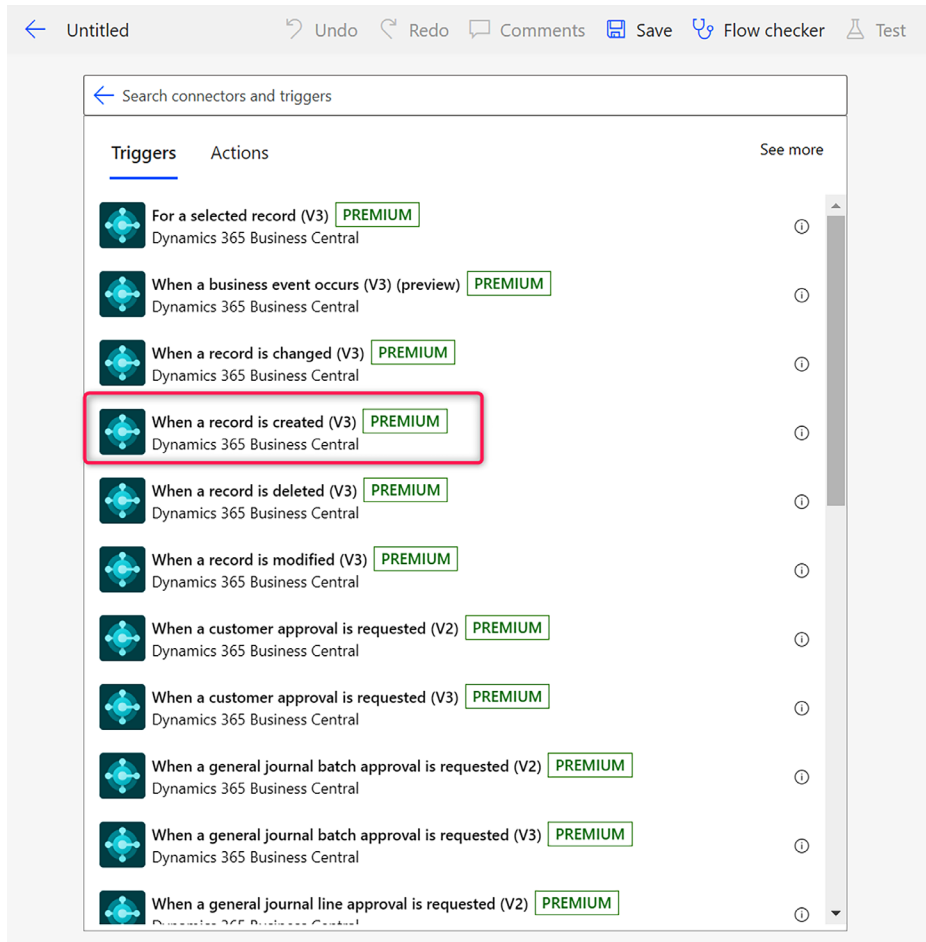


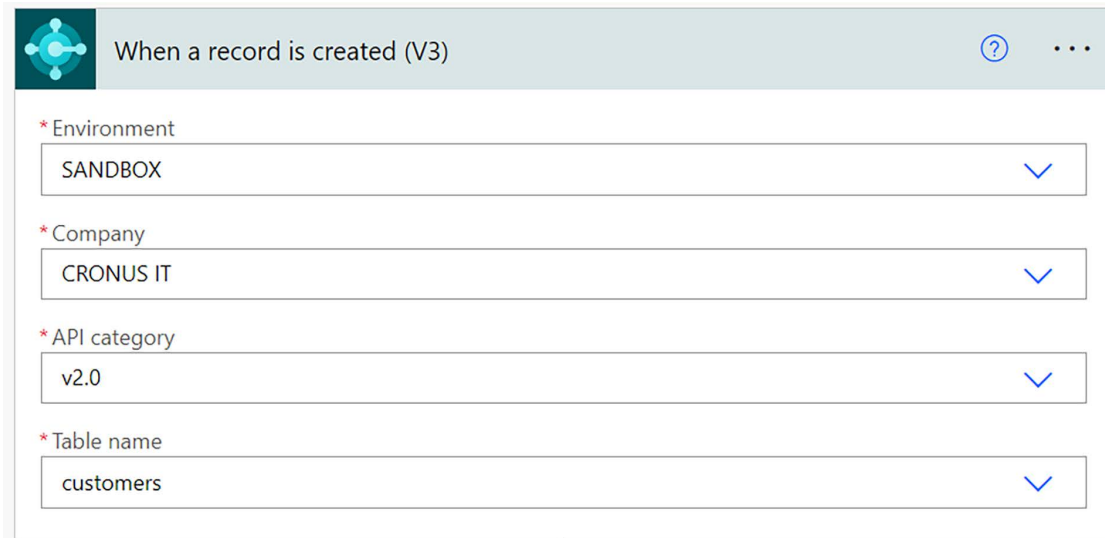
Figure 16.2: When a record is created (V3) trigger

This is the trigger that starts when a new record is created in Dynamics 365 Business Central.



Please note that triggers and actions in the Power Automate connector are always versioned for compatibility reasons when there are platform changes. In the Business Central connector for Power Automate, you need to use the new V3 actions now (V2 actions will be soon deprecated and removed).

When selected, we need to fill the **Environment** and **Company** parameters and then we need to select the entity on which to react (so, we need to select **API category = v2.0** and **Table name = customers**). Remember that the table name is always an API exposed from the ERP side (so, the **API category** can be the Microsoft's API version or your custom API version):

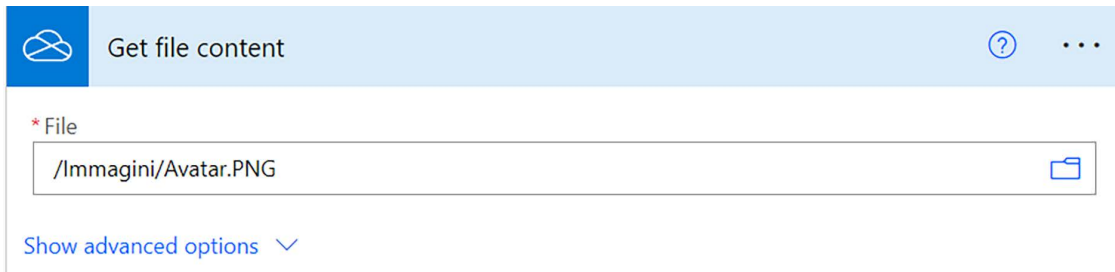


The screenshot shows the configuration for a trigger named "When a record is created (V3)". It features four required fields, each with a dropdown arrow:

- \* Environment**: Set to "SANDBOX".
- \* Company**: Set to "CRONUS IT".
- \* API category**: Set to "v2.0".
- \* Table name**: Set to "customers".

Figure 16.3: Configuring the When a record is created (V3) trigger

Now that we have defined the workflow trigger, we need to define the actions to execute. The first thing we need to do is retrieve the image from OneDrive, so we select the OneDrive connector and from it we select the **Get file content** action, providing the connection to OneDrive and the path of the image to retrieve:



The screenshot shows the configuration for the "Get file content" action. It features a single required field:

- \* File**: Set to "/Immagini/Avatar.PNG".

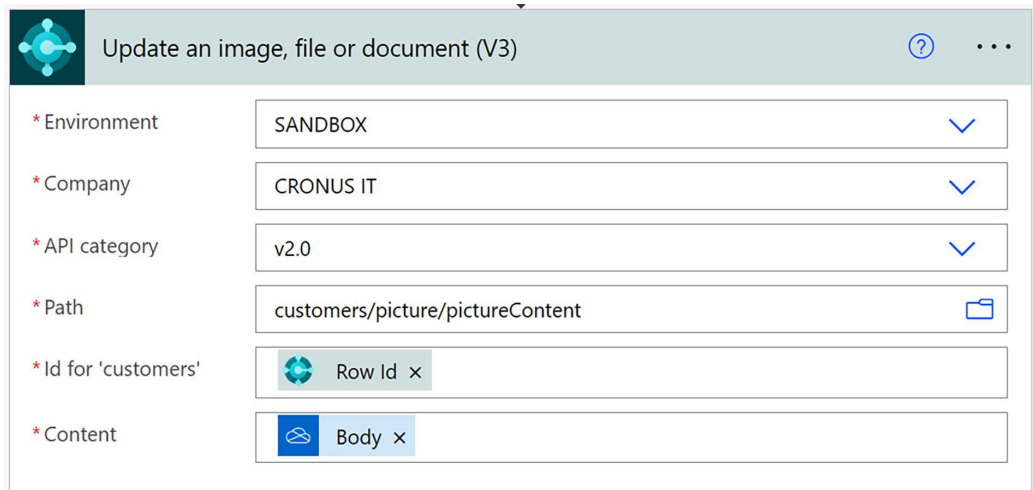
Below the field is a link labeled "Show advanced options" with a dropdown arrow.

Figure 16.4: Selecting the Get file content action

When we have the image, we need to add the image to the Customer record.

To do that, we select the Business Central connector and then select the **Update an image, file or document** action. Here:

1. Select the **Business Central environment and company**.
2. Select the **API category**.
3. In the **Path** parameter, select **customer** and then navigate until you reach the **pictureContent** entity.
4. In the **Id for 'customers'** parameter, select the **Row Id** coming from the **When a record is created** action (dynamics content).
5. In the **Content** parameter, select the **Body** parameter coming from the **Get file content** action of the OneDrive connector.



Update an image, file or document (V3)	
* Environment	SANDBOX
* Company	CRONUS IT
* API category	v2.0
* Path	customers/picture/pictureContent
* Id for 'customers'	Row Id
* Content	Body

Figure 16.5: Configuring the Content parameter

The final workflow will look like the following:

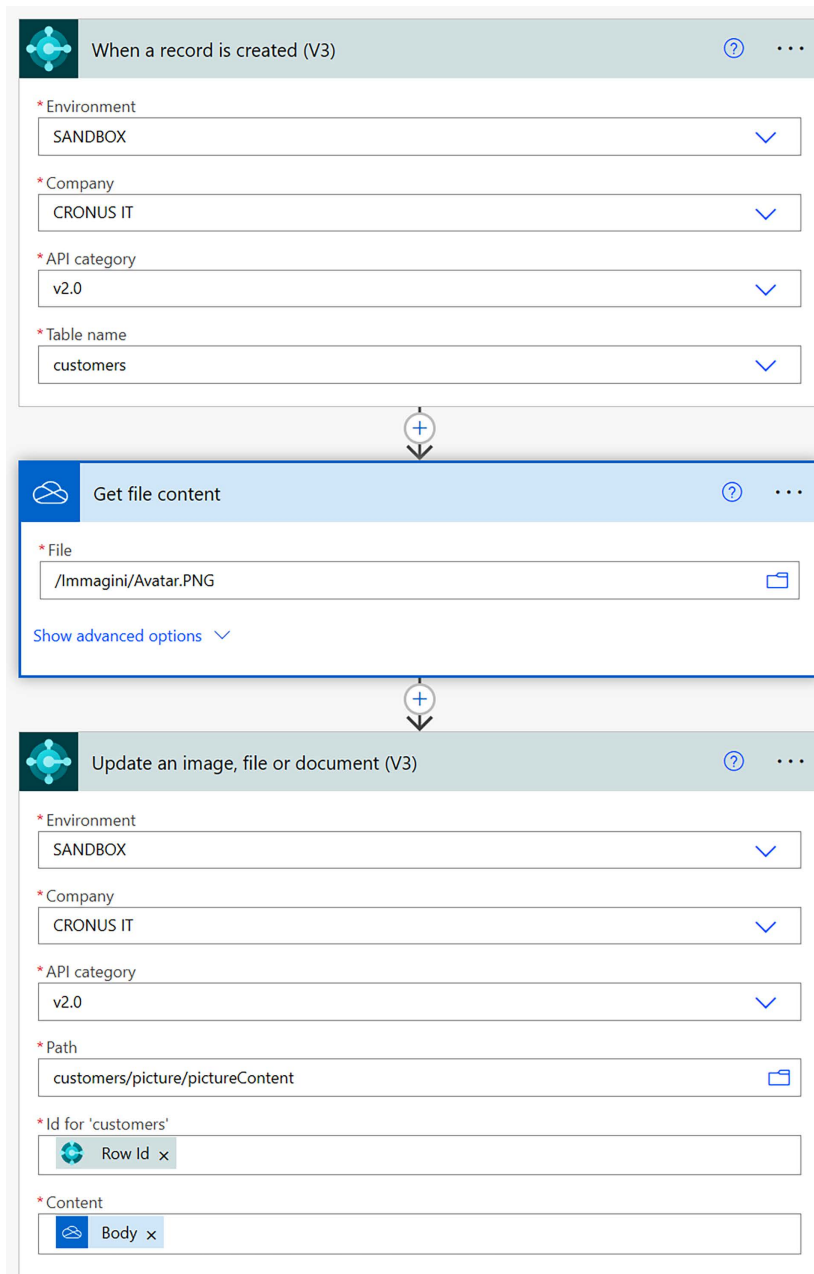


Figure 16.6: Completed workflow

Give the workflow a meaningful name and save it. Now, when someone creates a **Customer** record in Dynamics 365 Business Central, the workflow starts and a default image is associated with the newly created record:

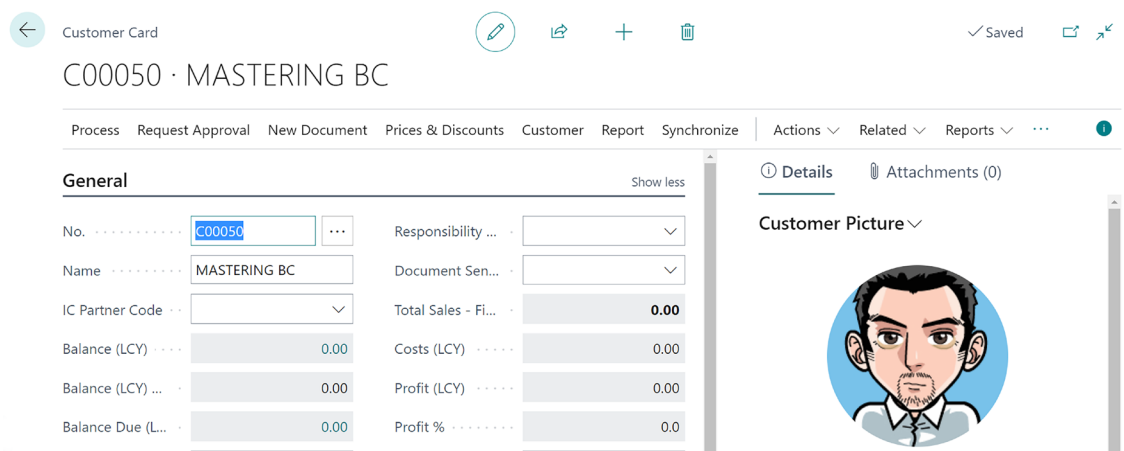


Figure 16.7: Viewing the flow from a Customer record

In the Power Automate portal, you can also check the history of the executions of a selected workflow and monitor failures:

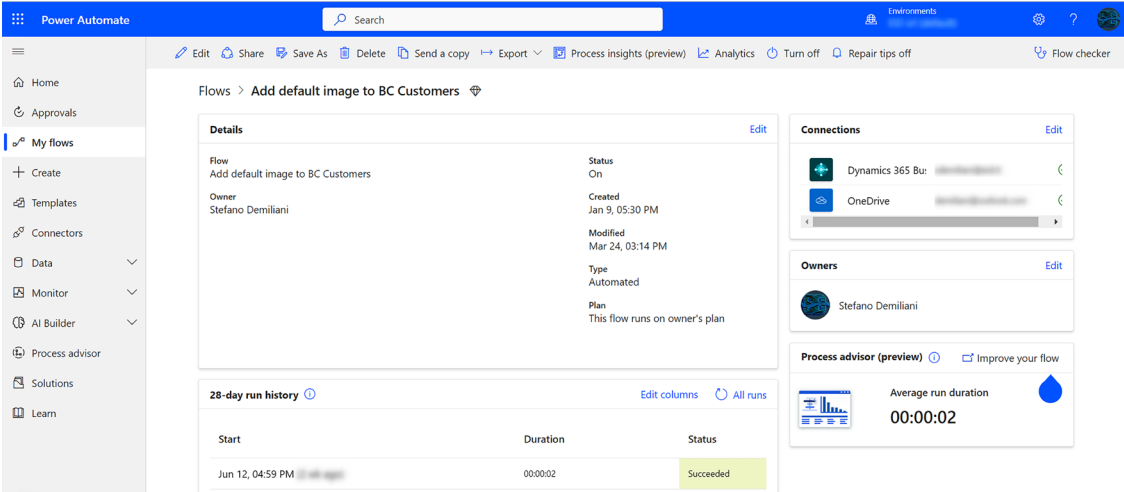
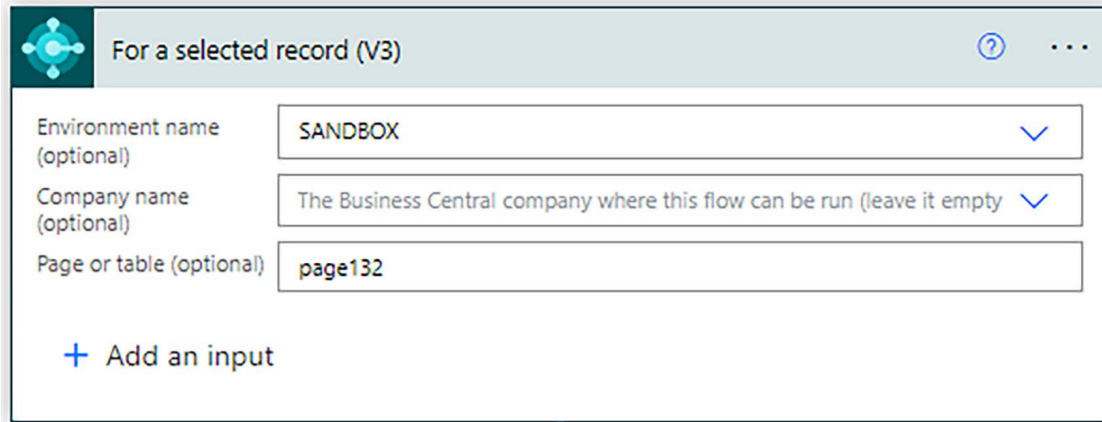


Figure 16.8: Monitoring flow execution

## Example 2: Exporting a selected invoice as PDF to OneDrive

This is a new and popular feature you will most likely find yourself using whenever your projects integrate Power Automate with Business Central. Dynamics 365 Business Central can execute Power Automate workflows in the context of the selected record by using the **For a selected record** trigger of the Business Central connector. When a flow starts with that trigger, it can automatically surface on the specified Business Central page or (if you specify a table) on all pages that have the selected tables as a source.

For this scenario, we use the **For a selected record (V3)** trigger and we select **page132 (Posted Sales Invoice page)** as the only source on which the workflow must be visible in the **user interface (UI)**. The workflow trigger will be as follows:



**For a selected record (V3)**

Environment name (optional): SANDBOX

Company name (optional): The Business Central company where this flow can be run (leave it empty)

Page or table (optional): page132

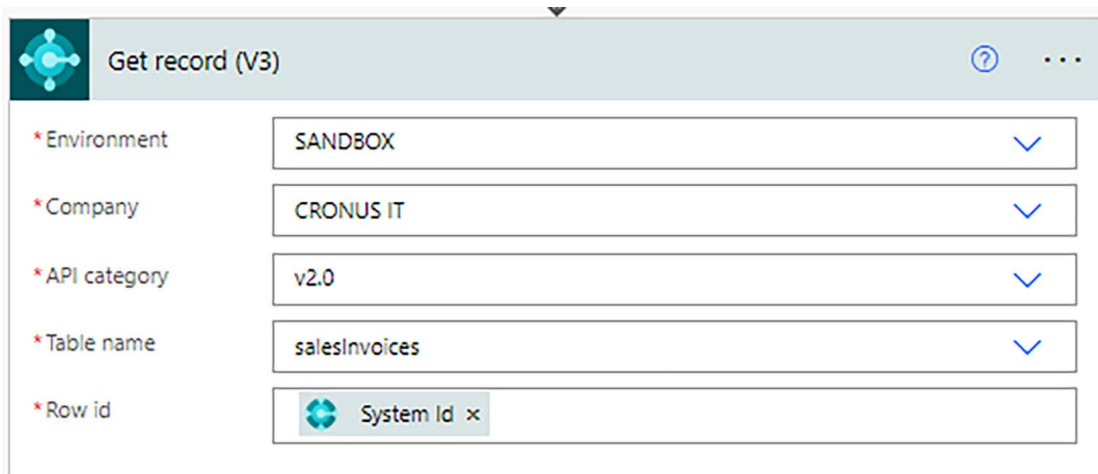
+ Add an input

Figure 16.9: Selecting the workflow trigger

As you can see from the trigger, the **Environment name** and **Company name** parameters are optional. This means that if you leave that blank, the workflow will be available for any environment and any company accordingly. In our example, we have filled the **Environment name** parameter and the **Company name** parameter is blank. This means that our workflow will be available in every company in the specified environment.

After defining the workflow trigger, we need to define the actions to execute. For the selected posted sales invoice record, we need to retrieve its reference, retrieve its PDF document content, and then save the PDF to OneDrive.

To retrieve a Dynamics 365 Business Central record, you can use the **Get record** action as follows:



**Get record (V3)**

\*Environment: SANDBOX

\*Company: CRONUS IT

\*API category: v2.0

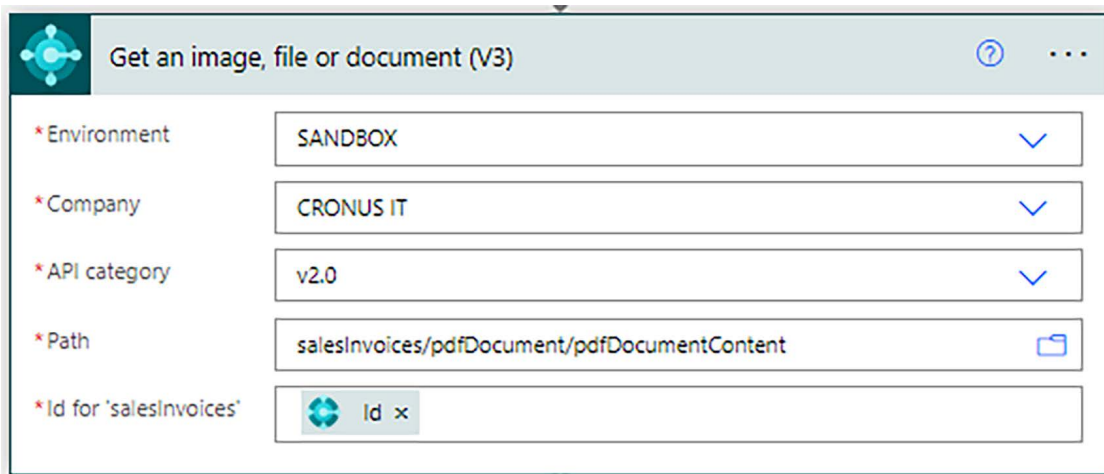
\*Table name: salesInvoices

\*Row id: System Id

Figure 16.10: Retrieving a record with Get record (V3)

Here, we select **salesInvoices** as **Table name** (API) and then we pass the **System Id** field (returned from the workflow trigger) as the **Row Id** parameter for retrieving the record.

When we have the record, to retrieve the PDF content of the selected document you can use the **Get an image, file or document** action as follows:



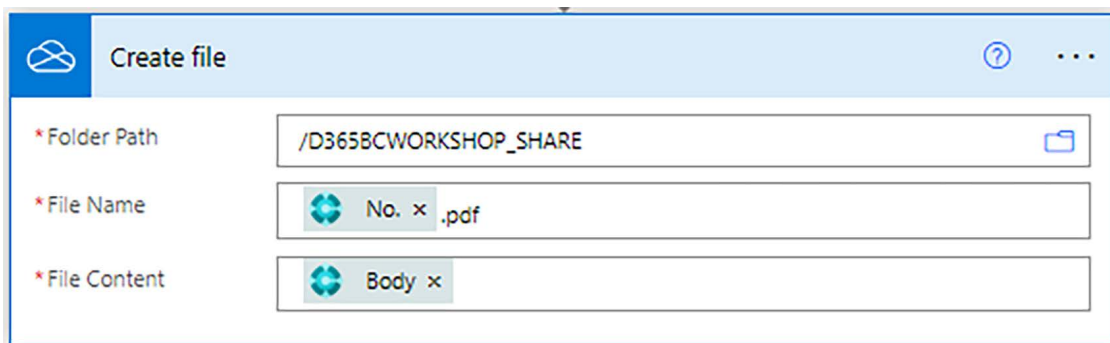
The screenshot shows the configuration for the 'Get an image, file or document (V3)' action. The title bar includes a connector icon, the action name, a help icon, and a menu icon. The configuration area contains five fields, each with a red asterisk indicating a required field:

- \*Environment**: A dropdown menu with 'SANDBOX' selected.
- \*Company**: A dropdown menu with 'CRONUS IT' selected.
- \*API category**: A dropdown menu with 'v2.0' selected.
- \*Path**: A text box containing 'salesInvoices/pdfDocument/pdfDocumentContent' with a folder icon on the right.
- \*Id for 'salesInvoices'**: A text box with a connector icon, the text 'Id', and a close icon 'x'.

Figure 16.11: Get an image, file or document action

This action wants the path of the pdfDocumentContent element for the selected record and wants the Id of the record (here, I'm passing the Id of the Get record action).

When you have the PDF content, you can store it in OneDrive by using the **Create file** action of the OneDrive connector as follows:



The screenshot shows the configuration for the 'Create file' action. The title bar includes a OneDrive connector icon, the action name, a help icon, and a menu icon. The configuration area contains three fields, each with a red asterisk indicating a required field:

- \*Folder Path**: A text box containing '/D365BCWORKSHOP\_SHARE' with a folder icon on the right.
- \*File Name**: A text box with a connector icon, the text 'No.', a close icon 'x', and the file extension '.pdf'.
- \*File Content**: A text box with a connector icon, the text 'Body', and a close icon 'x'.

Figure 16.12: Using the Create file action to store content

When saved, the workflow will surface on the specified page in Dynamics 365 Business Central (under the **Automate** action group):

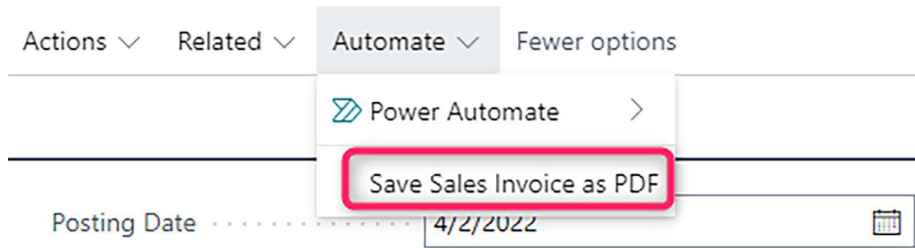


Figure 16.13: Viewing a flow from the Automate action group

Across the two examples in this section, we saw how you can use Power Automate in Dynamics 365 Business Central to automate tasks and create workflows.

In the next section, we'll see how to create a Power Apps solution connected to Dynamics 365 Business Central.

## Creating a Power Apps app with Dynamics 365 Business Central integration

In this section, we'll learn how to create a complex canvas app with Power Apps integrated with Dynamics 365 Business Central data.



Special thanks to Josh Anglesea for helping in writing this section. *Josh Anglesea* started working with NAV as an end user in 2010. He joined the Microsoft Partner community in 2013 and has held various roles from support consultant right up to solution architect. Josh is an active community member and enjoys writing blogs on Dynamics 365 Business Central and Power Platform. Josh has been a Microsoft MVP for Biz Apps since 2021. You can follow Josh on LinkedIn (<https://www.linkedin.com/in/josh-anglesea/>) or Twitter (@joshanglesea).

To provide an optimized UI, it is often advised to use Fluent UI components when developing Power Apps canvas apps. Details on how to get started with this approach can be found here: <https://learn.microsoft.com/en-us/power-platform/guidance/creator-kit/overview>. More details about Fluent UI can be found here: [https://developer.microsoft.com/en-us/fluentui#](https://developer.microsoft.com/en-us/fluentui#/).

The creator kit comes with a set of example canvas apps that showcase some of the capabilities. For instance, the **Creator Kit Reference App** has examples of each Fluent UI component in action. It allows a user to interact with the components but provides example code so you can make your own versions.

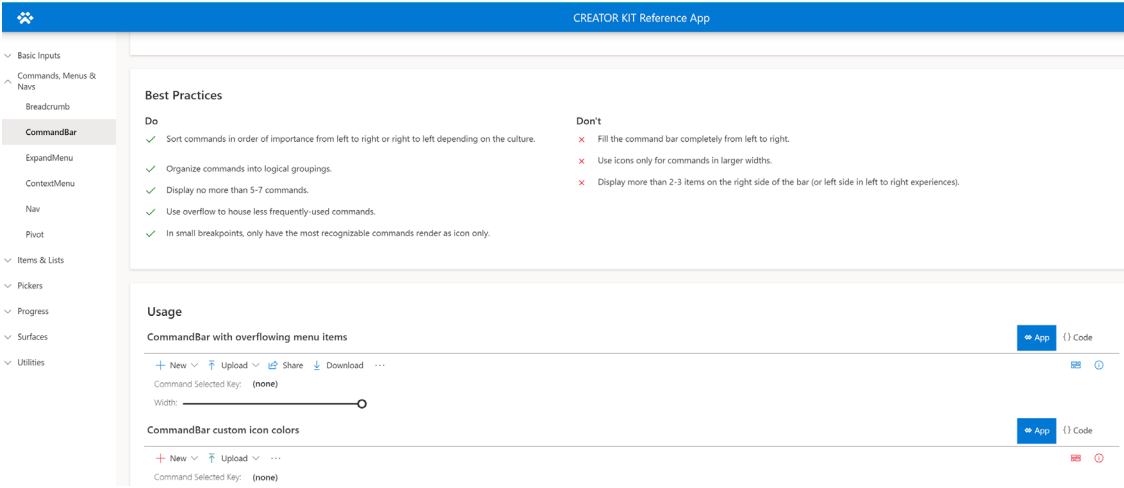


Figure 16.14: Canvas app creator kit

## Scenario: Expense app with offline capability

The aim of this scenario will be to build a canvas app for a phone. It must have the ability to locally save data in case the user is offline. The app will have a connection to Business Central tables so that the user can submit general journal lines. In this example, dimensions are being used to determine different types of expenses. However, in a production-based scenario, you may want to have individual G/L accounts for each expense type. This will ensure the correct application of tax amounts. Although the app will be built for a mobile device resolution, it can still be used in a browser.

Here is what you will need to get started:

- A connection to <https://make.powerapps.com/>
- A connection to Dynamics 365 Business Central (the example uses a cloud instance)
- A general journal batch ID for inserting the expenses in the required Business Central company. Here is an example of where to set up and retrieve this information (page inspection has been used here but you can gather the details from the Business Central API too):

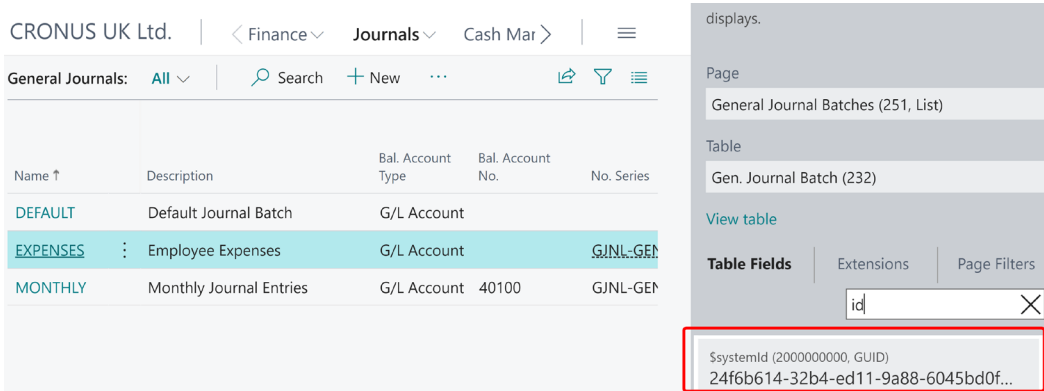


Figure 16.15: Locating a journal batch ID

- A dimension ID for a newly created dimension called **Expense-type**. Here is an example of where to set up and retrieve this information (page inspection has been used here but you can gather the details from the Business Central API too):

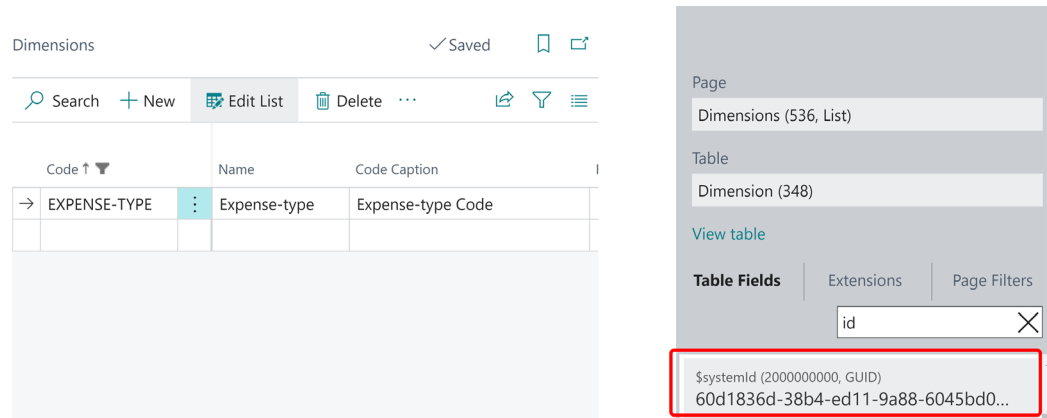


Figure 16.16: Locating a dimension ID

- Optionally, create a solution within Power Apps so that all components can be pulled together in one place. This is an optional step, but it follows ALM best practices:

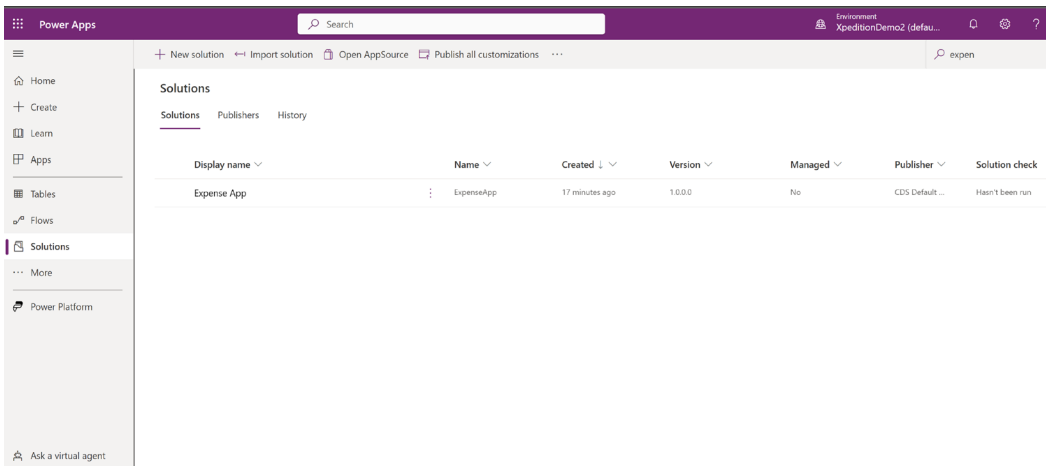


Figure 16.17: Solution for organizing components

Create two environment variables, one for the journal batch ID and one for the dimension ID:

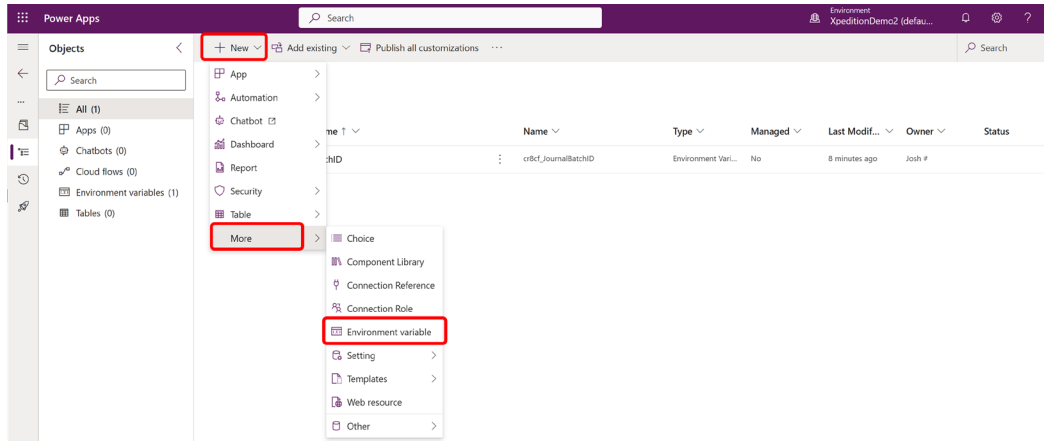


Figure 16.18: Navigation for creating environment variables

The example app will use standard API pages (available in v2.0 of the Business Central APIs). A full list can be seen in the documentation at the following link: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/api-reference/v2.0/>.

A list of the required API pages will be shown in the steps to build the app.

## Part 1: Creation of the canvas app project

To start creating the application, perform the following steps:

1. Open Power Apps Studio with your Office 365 account at <https://make.powerapps.com/>. There are different ways to build apps in Power Apps; we are going to use the canvas app from a blank template so that we can build an app for a mobile phone layout from scratch. Click on **Blank app**:

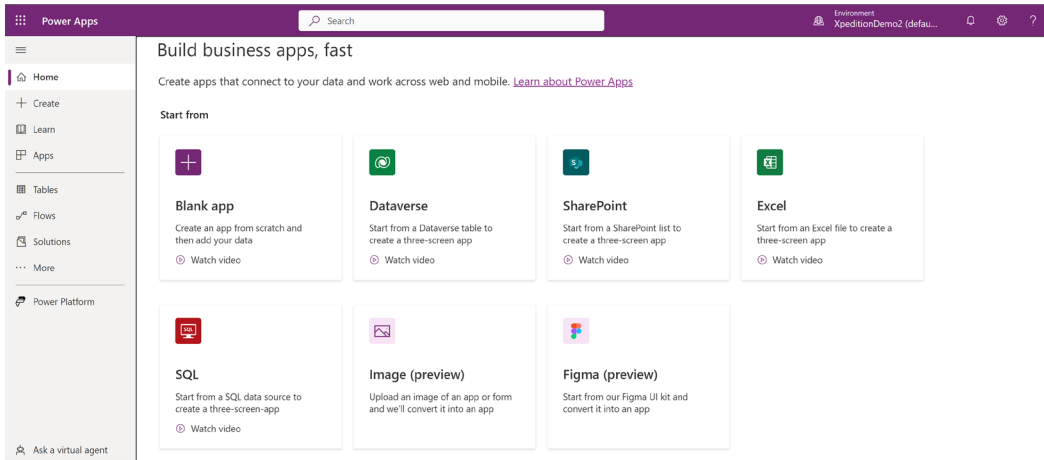


Figure 16.19: Creating a blank canvas app

2. Come up with a name for your app, select the format that you want (in this example, we will use the **Phone** format), and click on **Create**:

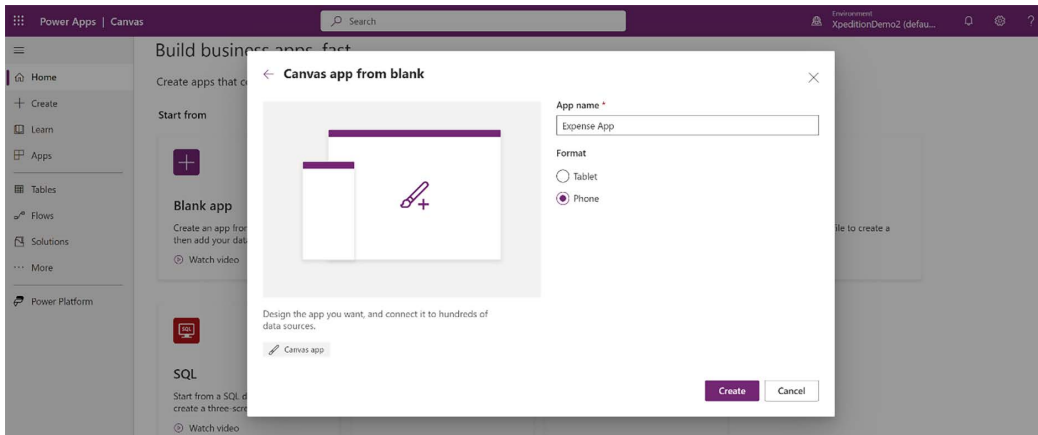


Figure 16.20: Selecting the Phone format

As soon as the application is created, you will be redirected to the canvas app designer, where we will start to build the components for our app.

## Part 2: Establishing a data connection to Dynamics 365 Business Central

We need to connect our app to Dynamics 365 Business Central by adding a data source in Power Apps using an existing connector. The connector concept here is the same as it is in Power Automate. In fact, Microsoft uses the same concept in all of the Power Platform applications. Let's continue the project we have worked on so far by applying this connector:

1. From either the side navigation menu, select **View** and then click on **Data Sources**. Or, choose the **Add Data** from the top navigation menu:

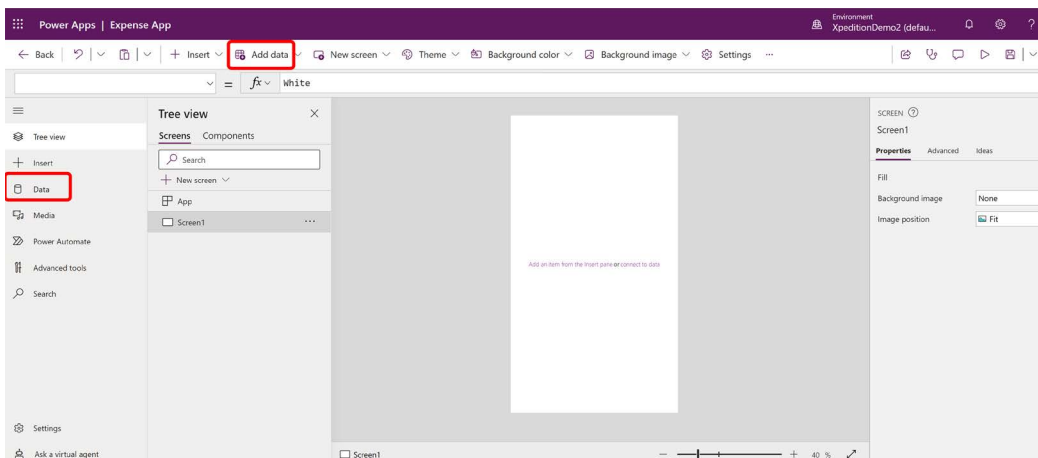


Figure 16.21: Option for adding a data source

- Now, add a data source. If you have already created a connection with Dynamics 365 Business Central (or other platforms) in any other Power Platform application, you can select the connection directly from the list that appears.
- If you have not created a connection with Dynamics 365 Business Central (or other platforms) in any other Power Platform application previously, click on **+ New connection**, search for **Business Central**, and select the Business Central connector, as follows:

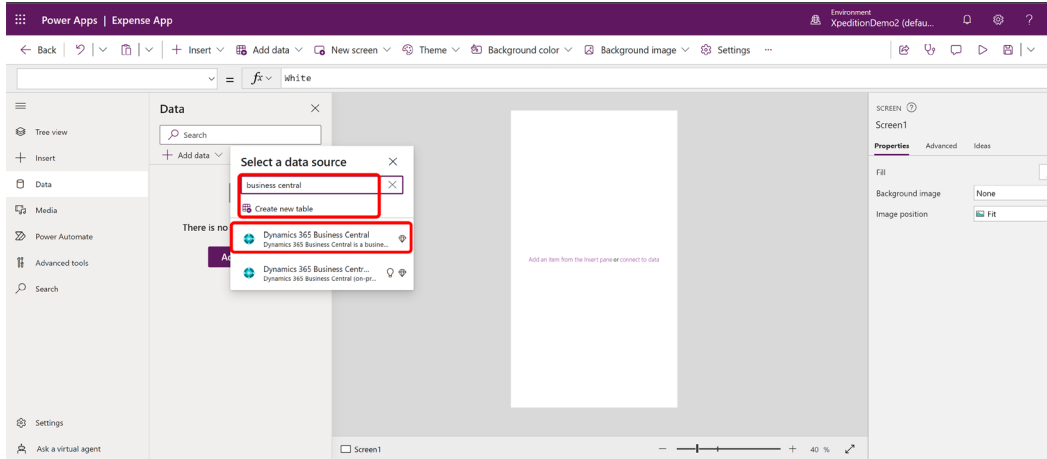


Figure 16.22: Selecting the Business Central connector

- Confirm the user account you are connecting with. If you do not have a previously set-up connection, you will be prompted to sign in. Now, click on **Create**. By creating the connection (or selecting an existing one) with Dynamics 365 Business Central, you need to select the dataset that you want to use:

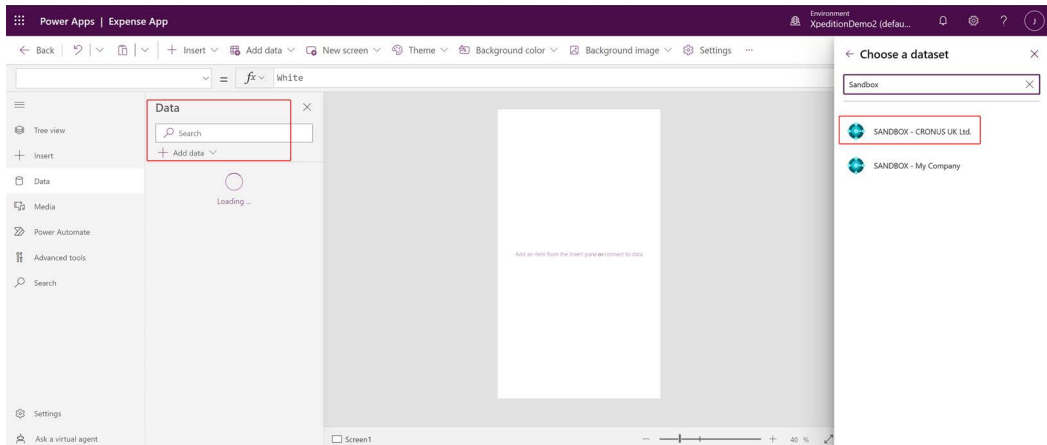


Figure 16.23: Selecting a dataset

## Part 3: Working with Dynamics 365 Business Central data from the canvas app

Now we can select which tables from Dynamics 365 Business Central we want to use in the app. Let's continue our project with this goal in mind:

1. In our example, we will select the **journals** table, the **journalLines** table, the **employees** table, and the **dimensionValues** table, and then click on **Connect**:

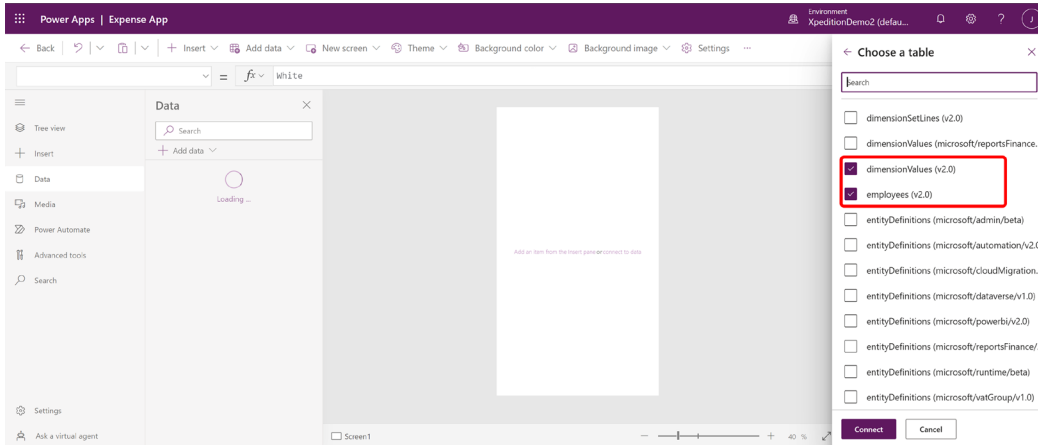


Figure 16.24: Selecting a data table

2. In addition to the Business Central data, if you have chosen to use a solution, you will need to add access to the environment variables. These are the additional data sources required to access the variables:

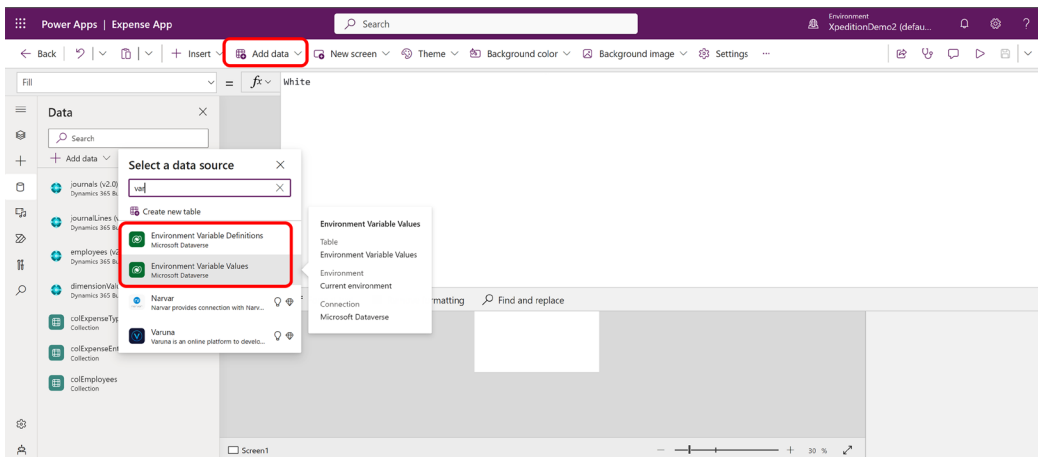


Figure 16.25: Additional solution data sources

3. The connection has been made. Now, we can start adding the controls and business logic to our app to consume and interact with those connections.

## Part 4: Adding controls and business logic to the canvas app

Components are used in Power Apps to ensure a consistent look and remove extra tasks from the developer where possible. In a Power Apps context, components are reusable controls that have properties that can be set per page. To practice using components, let's continue our project by adding three screens to the app:

1. Choose Components from the Tree view and create a new component:

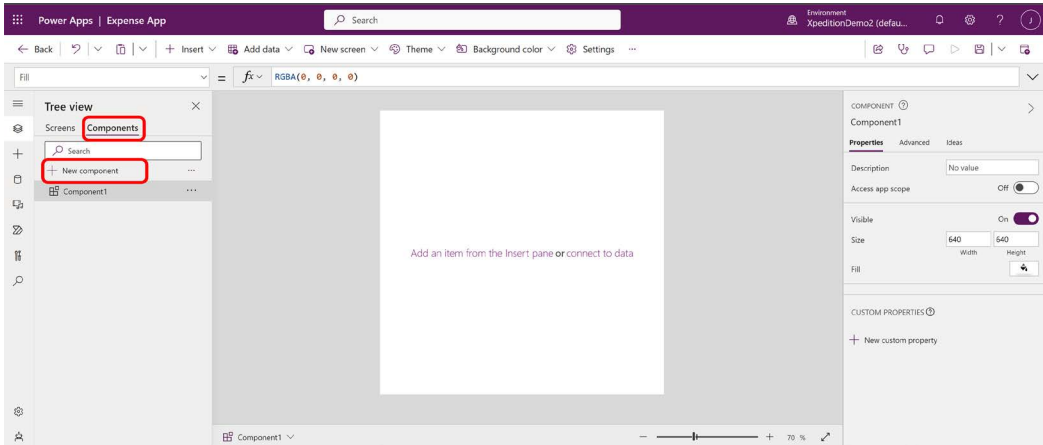


Figure 16.26: Creating a new component

2. Two components will be created: one for the header of the app and another for the menu navigation. Alter the Height to 100 pixels, the fill color, and add a custom property for the header text value:

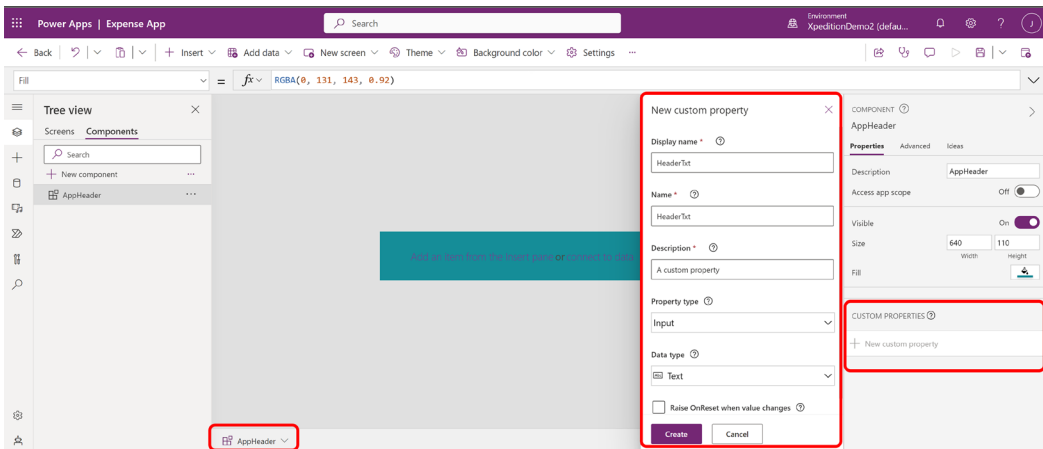


Figure 16.27: Altering the component properties

3. Add a **Text** label for displaying the custom property and set the **Text** property of the label to be that of the component **AppHeader** (which is the name of the example component):

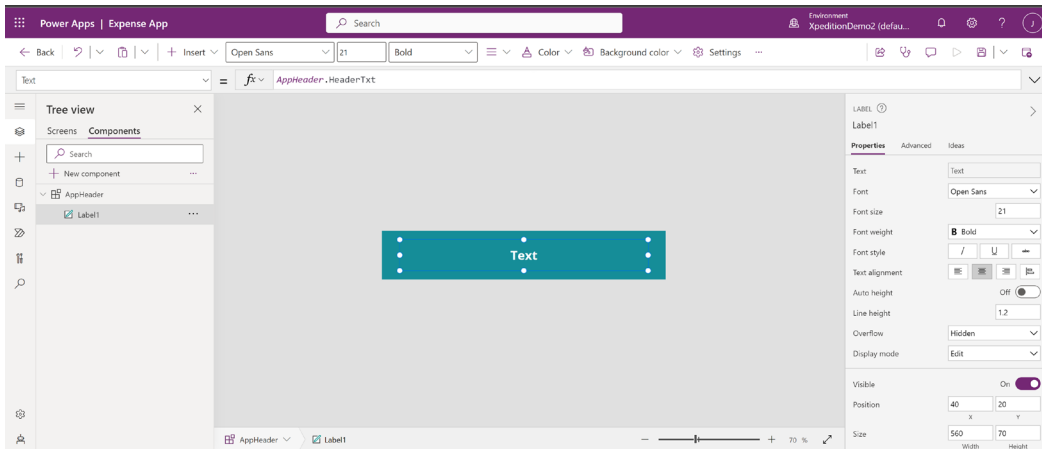


Figure 16.28: Configuring text properties

4. Create another component called **AppNavigation** and add a button in a contrasting color. Set the **Text** property of the button to be blank so the button is just a blank overlay on the darker background. Alter the **Display mode** property to be **View**. This will stop any interaction with the button control by the user:

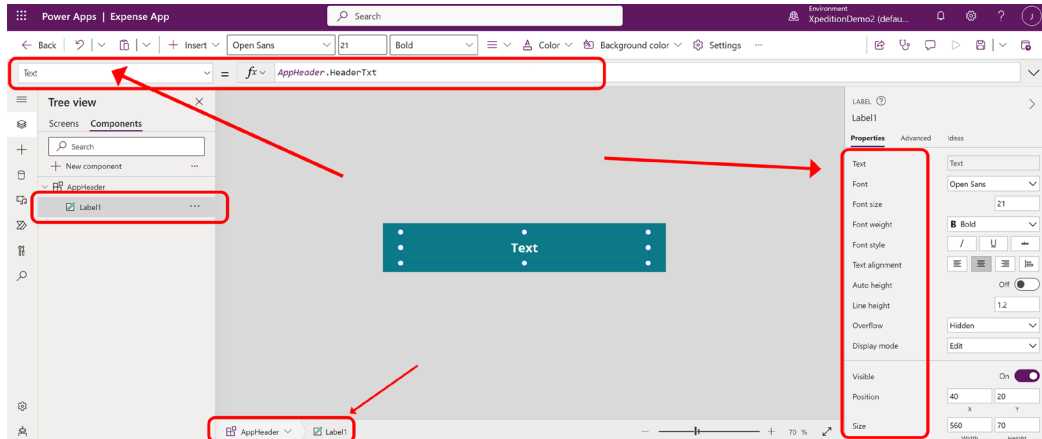


Figure 16.29: Configuring the second component's properties

5. Add icons, which will be used for navigating to different pages or performing certain actions in the app, and will therefore be the buttons we want a user to interact with:

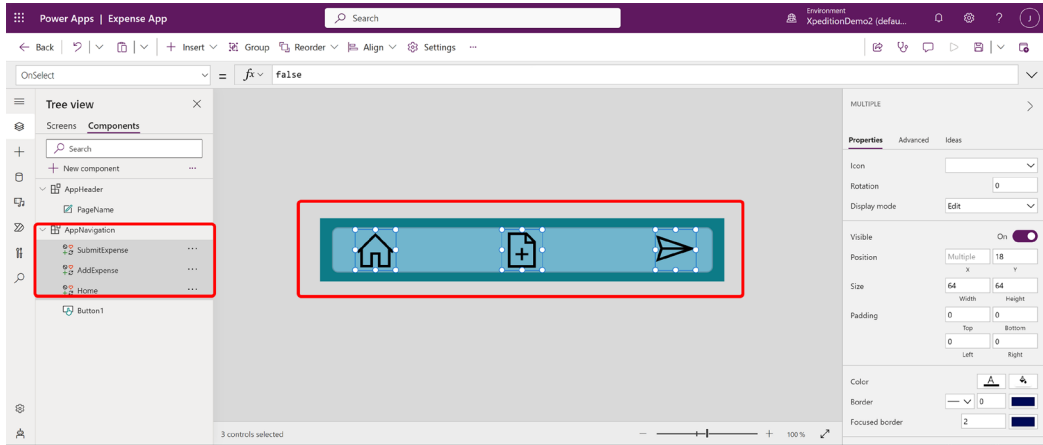


Figure 16.30: Adding navigation icons

6. Return to the **Screens** area of the **Tree view**. Rename Screen1 to **HomePage**. Using **Insert**, from the top navigation, add the components onto the page:

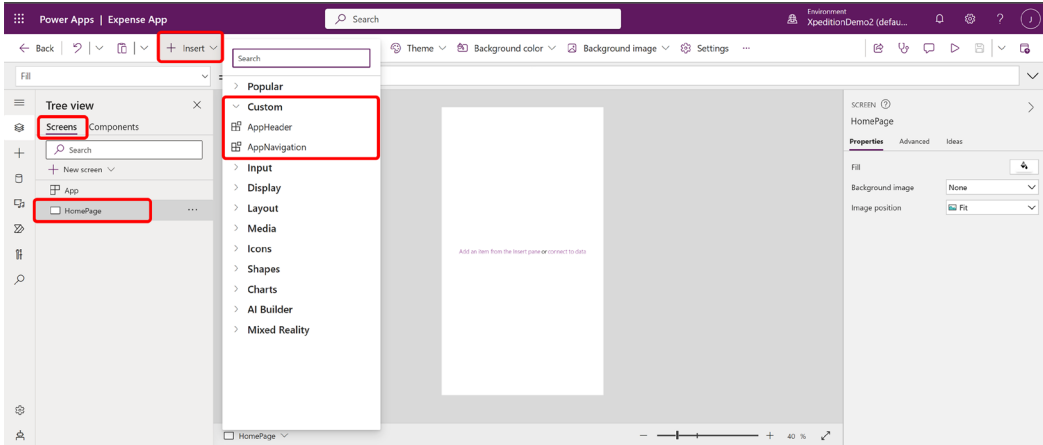


Figure 16.31: Adding the components to the page

- Once added, select **AppHeader** and alter the **HeaderText** custom property by adding the following Power Fx formula:

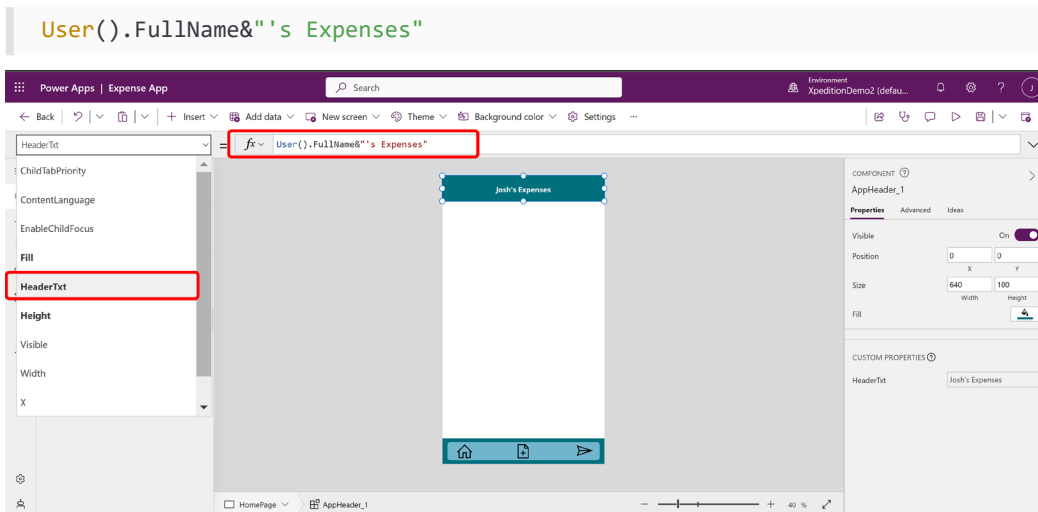


Figure 16.32: Adding a PowerFX formula

- Add another screen to the app using the **Blank** option:

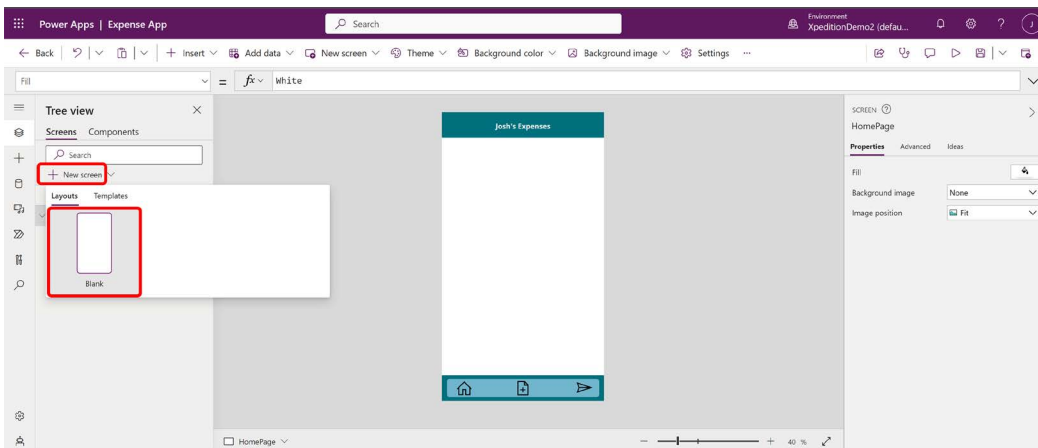


Figure 16.33: Adding a new blank screen

9. Call the new screen **ExpenseEntry** and add the same two components to the page. Ensure that you update the **HeaderText** value:

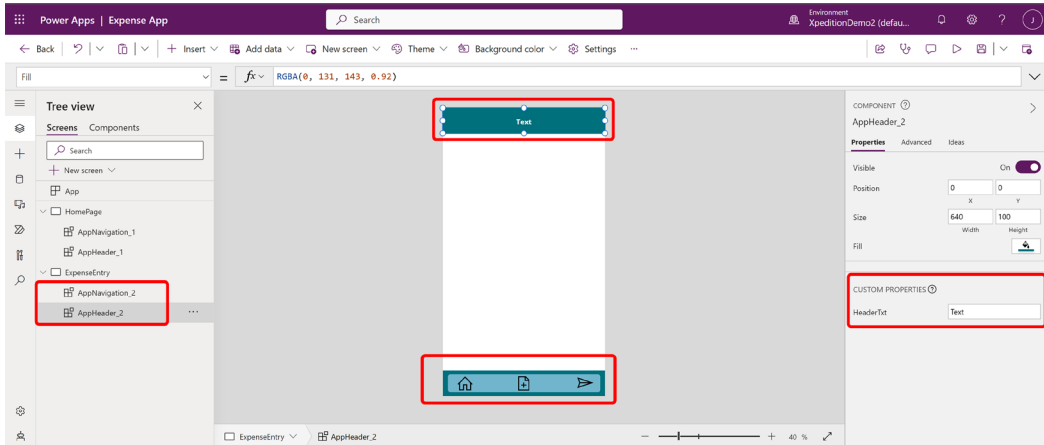


Figure 16.34: Adding components to the ExpenseEntry screen

10. Note that because we have used components, if we now decide we want the color of the header to be different, we only need to make this change in the **Components** area. Any screen or page that uses the component will then receive the update.
11. Back on the home page, add two additional icons and place them centrally on the screen. Add two text labels to provide context for what the icons will give the user. Ensure to rename the controls as they are added to the app. This will allow better context when reviewing Power Fx formulas or with app debugging:

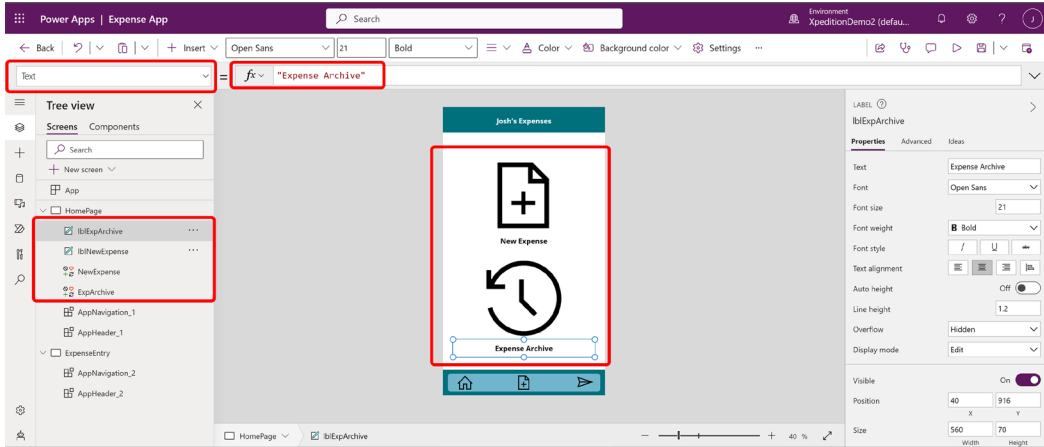


Figure 16.35: Adding icons to the home page

12. With the **NewExpense** icon, change the **OnSelect** property so it navigates a user to the **ExpenseEntry** screen:

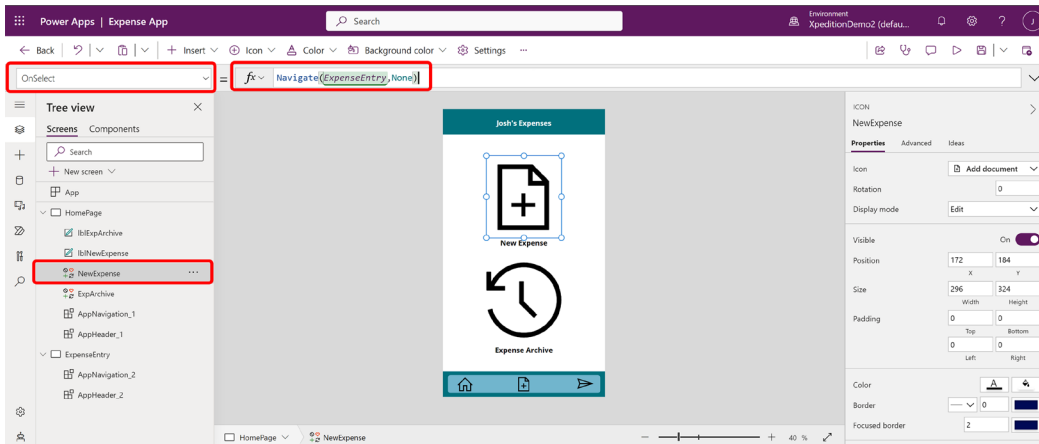


Figure 16.36: Adding logic to the NewExpense icon

13. The icons in the **AppNavigation** component need to be set within the component itself. This means that it will interact the same no matter where it is present in the app. Go back to the **Components** area of the **Tree view** and alter the **OnSelect** property for the **Home** and **AddExpense** icons (the **SubmitExpense** icon will be covered later):

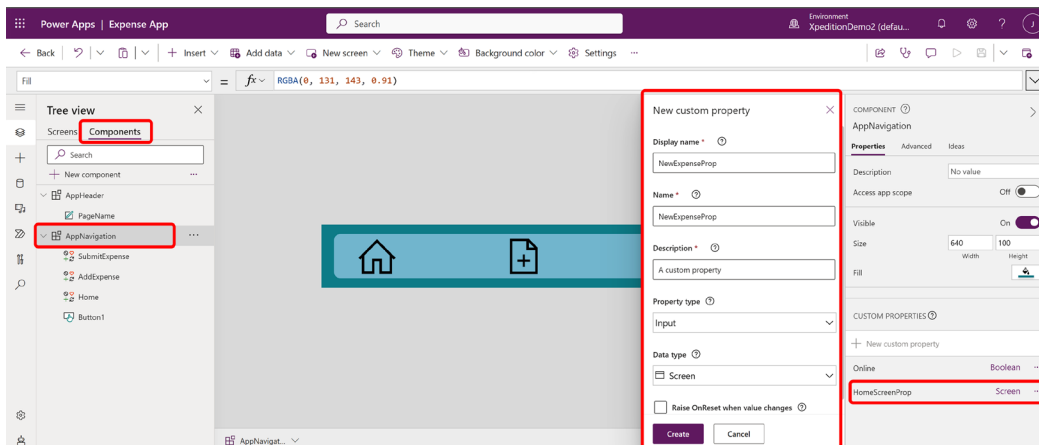


Figure 16.37: Adding logic for the Home and AddExpense icons

14. Now, set each corresponding icon to use the custom property within their **OnSelect** property with the displayed Power Fx formula:

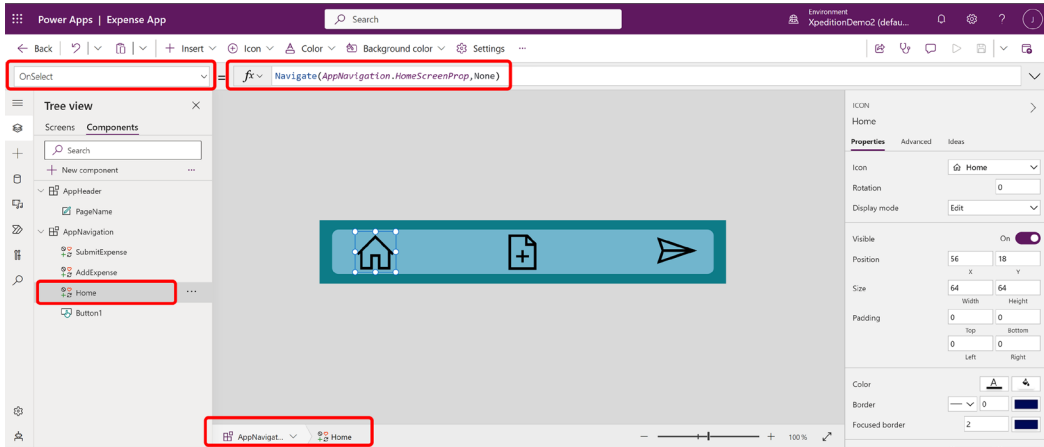


Figure 16.38: Configuring each icon

15. On each screen, set the custom properties for the AppNavigation component by adding the name of each page that the icon needs to use:

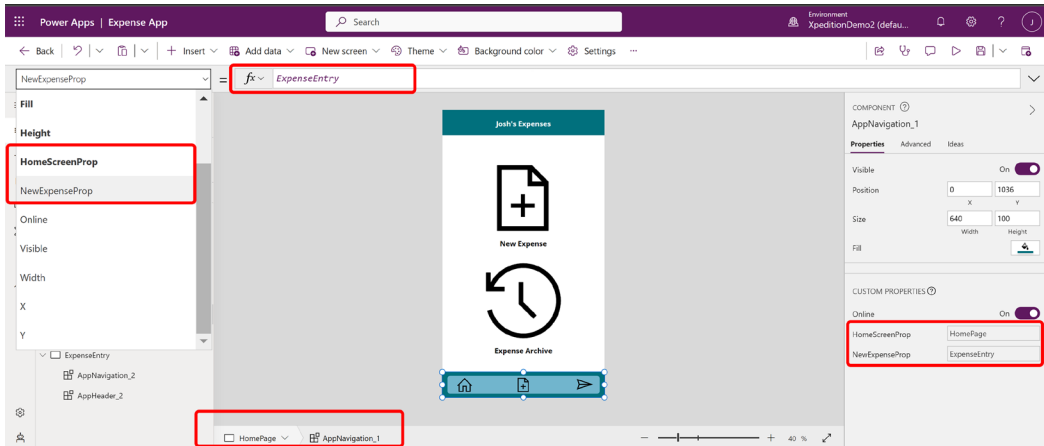


Figure 16.39: Setting component properties with page names

16. Go to the **ExpenseEntry** screen and add the following controls: date picker, dropdown, and two text inputs. Rename each one so that they have a valid context for use in Power Fx formulas:

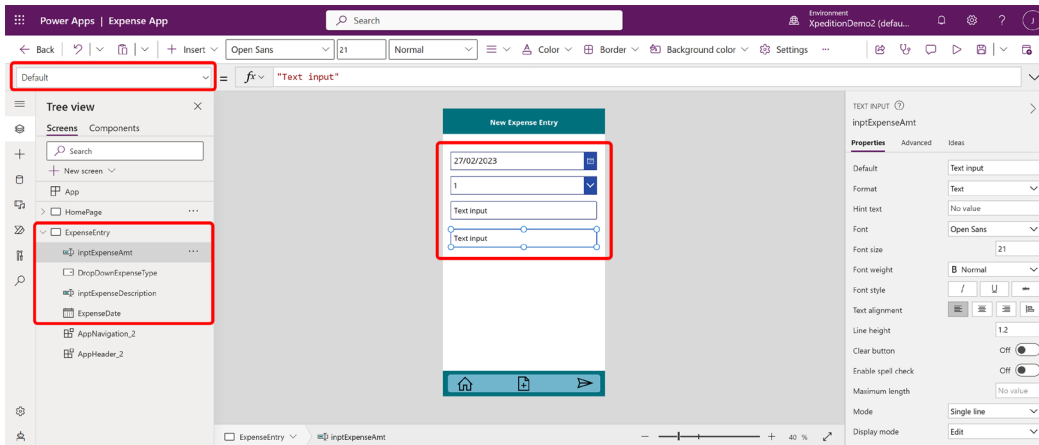


Figure 16.40: Adding controls to the ExpenseEntry screen

17. With the text inputs, alter the Default property to "" so it displays a blank value. Alter the HintText property for both text inputs to the following:

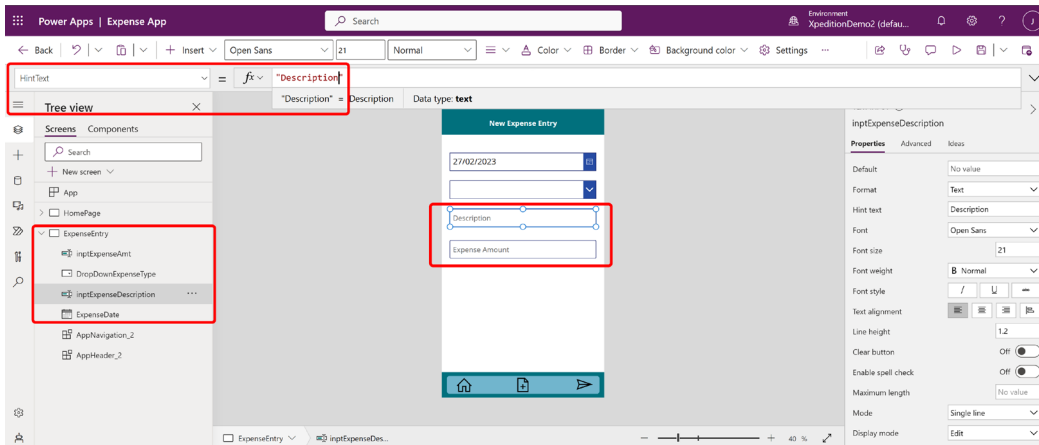


Figure 16.41: Setting the text inputs

Our canvas app UI is now customized with the layout we want. Now, we'll start adding support for offline capabilities in the app.

## Part 5: Adding offline capabilities to the application

We now have a canvas app connected to Business Central data, complete with screens and actions for users to navigate. Great work! Now, we can take our app further by implementing a popular Power Apps capability – working offline:

1. Having an app work offline requires the user to interact with data that is stored in collections. These collections can be saved locally with a canvas app installed on the device. Let's establish this logic so that we can move between online and offline accordingly. Choose the OnStart property of the app from the side navigation bar. A prerequisite with this example is that you have determined a general journal batch you will use and created a new dimension for expense types.

Insert the following Power Fx formula and click **Format text** after so that the formula is more legible:

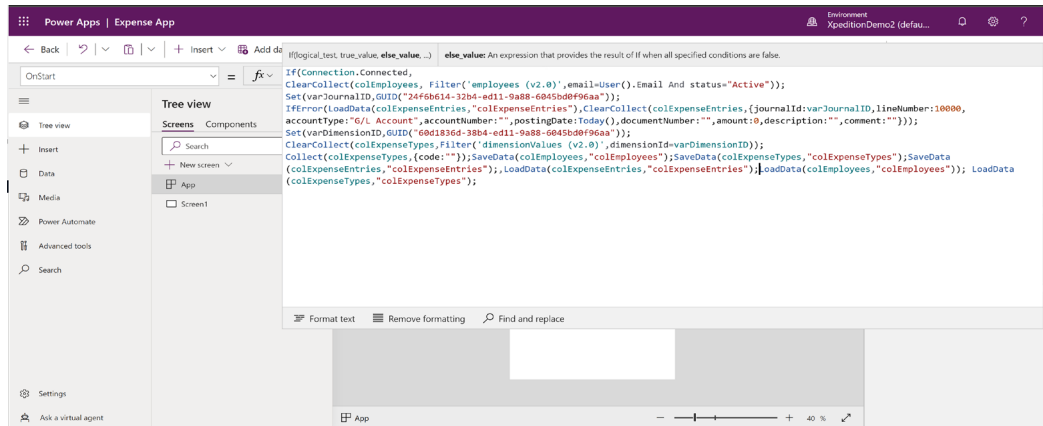


Figure 16.42: Power Fx formula for offline capability

Here's the Power Fx formula split into smaller parts:

```
If(
    Connection.Connected,
    ClearCollect(
        colEmployees,
        Filter(
            'employees (v2.0)',
            email = User().Email And status = "Active" And number <>" "
        )
    );
    Set(
        varJournalID,
        GUID("24f6b614-32b4-ed11-9a88-6045bd0f96aa")
    );
    //An attempt will be made to Load data from the local device first from a
    previous session
    IfError(
        LoadData(
            colExpenseEntries,
```

```

        "colExpenseEntries"
    ),
    //This is a blank entry to the expense entry collection - which is the
    //online/offline local storage for the data. If the user is not connected
    //there is a need to start the collection so that the expected fields are
    //known
    ClearCollect(
        colExpenseEntries,
        {
            journalId: varJournalID,
            accountType: "G/L Account",
            accountNumber: "10000",
            postingDate: Today(),
            amount: 0,
            description: "",
            comment: ""
        }
    )
);
Set(
    varDimensionID,
    GUID("60d1836d-38b4-ed11-9a88-6045bd0f96aa")
);
//If online the previously setup dimension from Business Central will be
//retrieved directly from Business Central and stored in a local collection
ClearCollect(
    colExpenseTypes,
    Filter(
        'dimensionValues (v2.0)',
        dimensionId = varDimensionID
    )
);
//Business Central will only provide real data. So that we can force the
//user to make an expense type selection later in the app, we place a blank
//value in the collection. Note, that this only happens in the scenario we
//are online, and the data is freshly retrieved from Business Central
Collect(
    colExpenseTypes,
    {code: ""}
);
//Save copies of the collections. This will allow for the data to be
//retrieved if the device falls offline or is booted up whilst offline

```

```
SaveData(  
    colEmployees,  
    "colEmployees"  
);  
SaveData(  
    colExpenseTypes,  
    "colExpenseTypes"  
);  
SaveData(  
    colExpenseEntries,  
    "colExpenseEntries"  
);  
,  
Set(  
    varJournalID,  
    GUID("24f6b614-32b4-ed11-9a88-6045bd0f96aa")  
);  
Set(  
    varDimensionID,  
    GUID("60d1836d-38b4-ed11-9a88-6045bd0f96aa")  
);  
LoadData(  
    colExpenseEntries,  
    "colExpenseEntries"  
);  
LoadData(  
    colEmployees,  
    "colEmployees"  
)  
);  
LoadData(  
    colExpenseTypes,  
    "colExpenseTypes"  
);  
Set(  
    varHomePage,  
    HomePage  
);
```

The logic being used will gather details directly from Business Central if connected. If not, it will use the `LoadData()` function to pull it from the local device. Note that if connected, the `SaveData()` function is used so that the data is in place if the user falls offline during usage, or boots up the app while offline next time. This adds a bonus feature to the app, allowing us to store the history of expenses submitted without having to retrieve posted entries from Business Central.

Note that the previously set up general journal batch and dimension are being directly set as variables for use in the app. This sort of hardcoding is being used in the absence of having a dynamic place to hold the reference. In reality, the best practice would be to environment variables.

Environment variables can be established within solutions and then retrieved within Power Apps or Power Automate. Given that this app needs to work offline, the environment variables have been skipped. However, this is how the Power Fx formula would look if they were in place. Check the highlighted text:

```
If(Connection.Connected,
ClearCollect(colEmployees, Filter('employees (v2.0)',email=User().Email
And status="Active" And number <>""));
Set(varJournalID,GUID(LookUp('Environment Variable Definitions','Schema
Name' = "cr8cf_JournalBatchID").'Default Value'));
IfError(LoadData(colExpenseEntries,"colExpenseEntries"),ClearCollect
(colExpenseEntries,{journalId:varJournalID,lineNumber:10000,accountType:"
G/L
Account",accountNumber:10000,postingDate:Today(),amount:0,description:"",
comment:""}));
Set(varDimensionID,GUID(LookUp('Environment Variable Definitions',
'Schema Name' = "cr8cf_DimensionID").'Default Value'));
ClearCollect(colExpenseTypes,Filter('dimensionValues (v2.0)'
,dimensionId=varDimensionID));
Collect(colExpenseTypes,{code:""});SaveData(colEmployees,"colEmployees");
SaveData(colExpenseTypes,"colExpenseTypes");SaveData(colExpenseEntries,
"colExpenseEntries");Set(varJournalID,GUID
("24f6b614-32b4-ed11-9a88-6045bd0f96aa"));Set(varDimensionID,GUID
("60d1836d-38b4-ed11-9a88-6045bd0f96aa"));LoadData
(colExpenseEntries,"colExpenseEntries");LoadData
(colEmployees,"colEmployees");
LoadData(colExpenseTypes,"colExpenseTypes");
```

2. Now that the collections have been established, it is possible to make use of them on other screens. The first place to set this is with the dropdown for expense types. Once you have altered the **Items** property, set the **Default** property of the dropdown to be blank. Note that the Power Fx formula from *Step 17* has a formula that adds a blank **ExpenseType** to the collection:

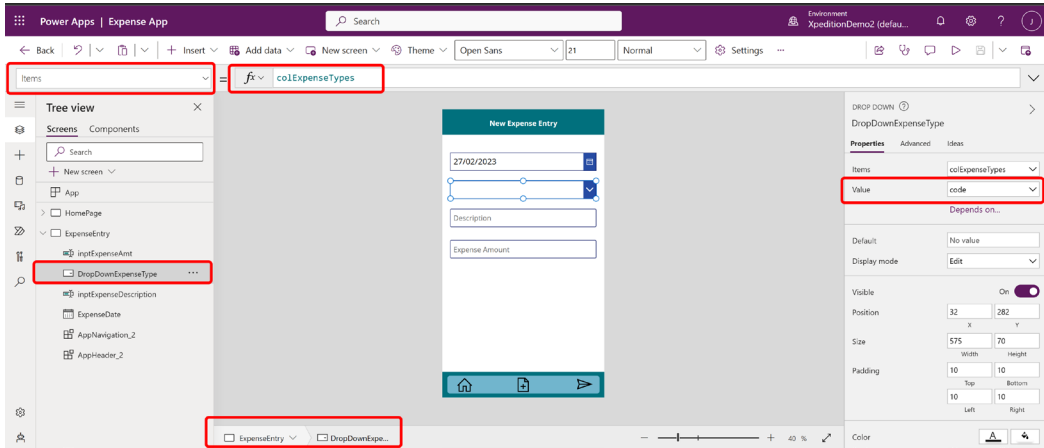


Figure 16.43: Applying the formula to set expense types

3. Alter the sorting of the drop-down box so that the blank entry, which was added in the **OnStart** Power Fx formula, is at the top of the list and the **Default** value is "". This is required so that we can add logic to ensure a user makes a relevant selection:

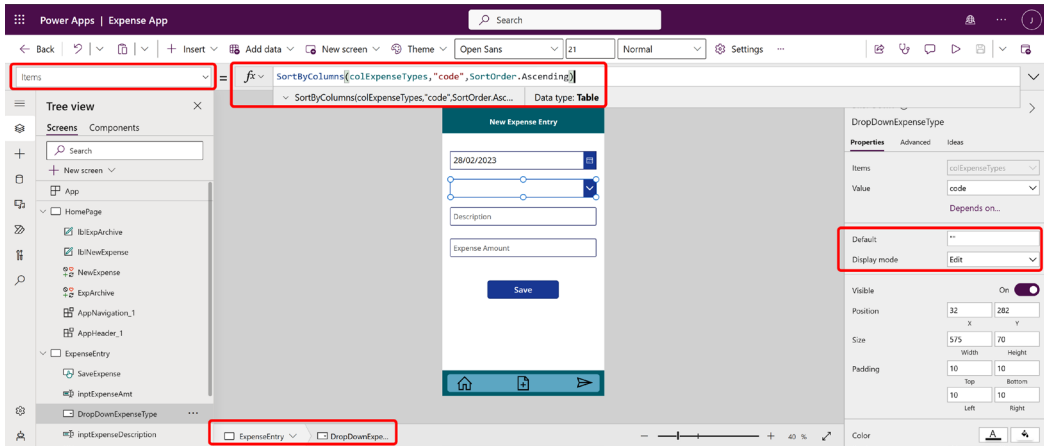


Figure 16.44: Arranging the drop-down box in Properties

4. Add a new button to the ExpenseEntry screen, making sure to rename it and add the text value **Save**. Within the **OnSelect** property of the **Save** button, add the following Power Fx formula. You will notice that the **Collect()** function is used twice. This is so that the journal entries balance in Business Central. This is required so that the general journal can be posted by a user in Business Central. For the app, a user does not need to know this is taking place. We will only want the user to see the values that they have inputted. The Power Fx formula shown in the following screenshot also has the functions to save the data to the local device if offline and to reset all the screen controls that data is retrieved from:

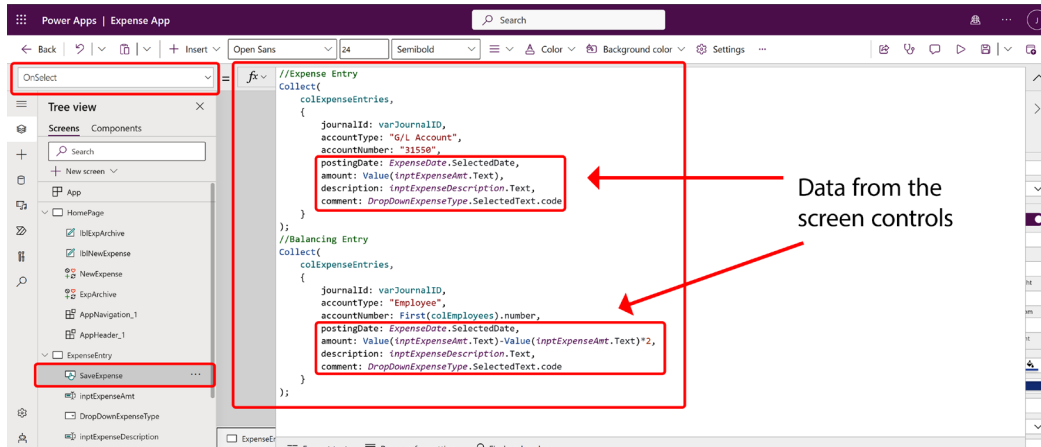


Figure 16. 45: Adding offline saving functionality

For easier readability, here's the Power Fx formula:

```
//Expense Entry
Collect(
    colExpenseEntries,
    {
        journalId: varJournalID,
        accountType: "G/L Account",
        accountNumber: "31550",
        postingDate: ExpenseDate.SelectedDate,
        amount: Value(inptExpenseAmt.Text),
        description: inptExpenseDescription.Text,
        comment: DropDownExpenseType.SelectedText.code
    }
);
//Balancing Entry
Collect(
    colExpenseEntries,
    {
        journalId: varJournalID,
```

```

        accountType: "Employee",
        accountNumber: First(colEmployees).number,
        postingDate: ExpenseDate.SelectedDate,
        amount: Value(inptExpenseAmt.Text) - Value(inptExpenseAmt.Text)*2,
        description: inptExpenseDescription.Text,
        comment: DropDownExpenseType.SelectedText.code
    }
);
If(
    Connection.Connected = false,
    SaveData(
        colExpenseEntries,
        "colExpenseEntries"
    )
);
Reset(ExpenseDate);
Reset(inptExpenseAmt);
Reset(inptExpenseDescription);
Reset(DropDownExpenseType)

```

5. To ensure a user has completed each field as we require, the Visible property of the Save button will be altered to check each field. If all fields are populated, then the button will be available to the user for selection. To do this, the following Power Fx formula has been added:

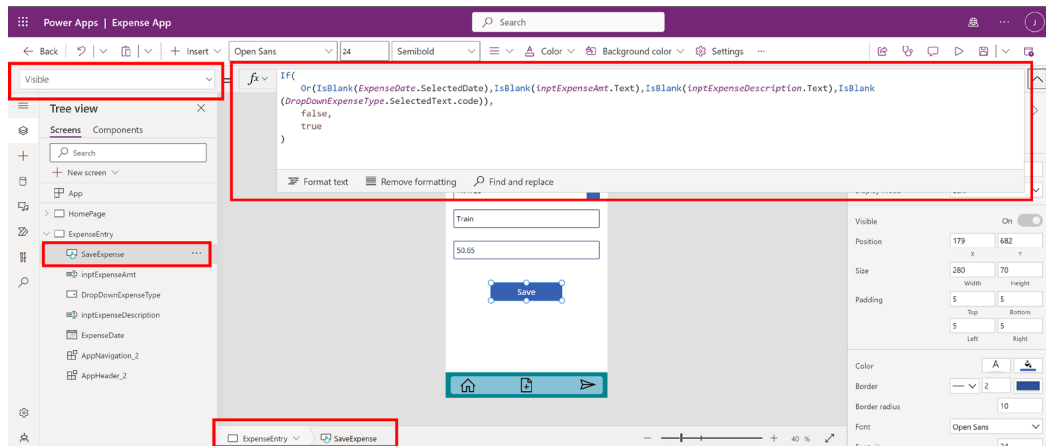


Figure 16.46: Adding functionality to the Save button for checking whether a field is populated

Here's the Power Fx formula we are using:

```

If(
    Or(IsBlank(ExpenseDate.SelectedDate), IsBlank(inptExpenseAmt.
    Text), IsBlank(inptExpenseDescription.Text), IsBlank(DropDownExpenseType.

```

```
SelectedText.code)),
    false,
    true
)
```

6. Create a new blank screen and call this **ExpenseArchive**. Add the components for the header and the navigation. Ensure that you alter the custom properties so they have relevant values:

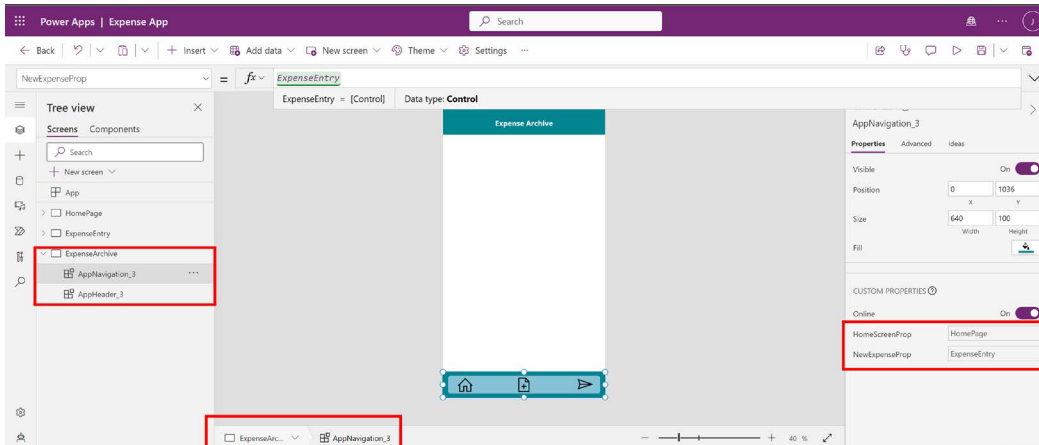


Figure 16.47: Adding the ExpenseArchive screen

7. Add a vertical gallery to the screen and assign the `colExpenseEntries` collection to the gallery. Ensure that you name the gallery something sensible:

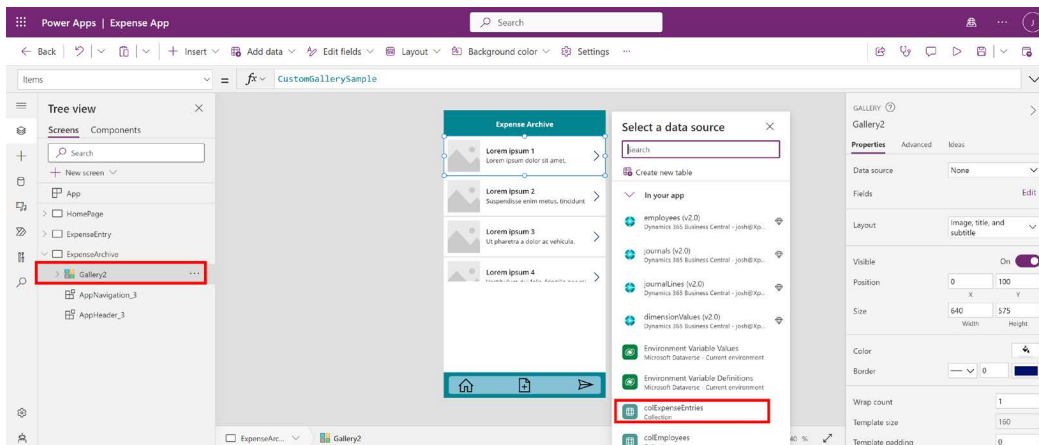


Figure 16.48: Adding a gallery to the Expense Archive

8. Change the gallery layout so that it uses the **Title, subtitle and body** option. Choose the **Edit** button under the **Fields** property of the gallery to select the fields you want to be displayed. Note that the **Items** property of the gallery has been filtered so that the user only sees the positive values:

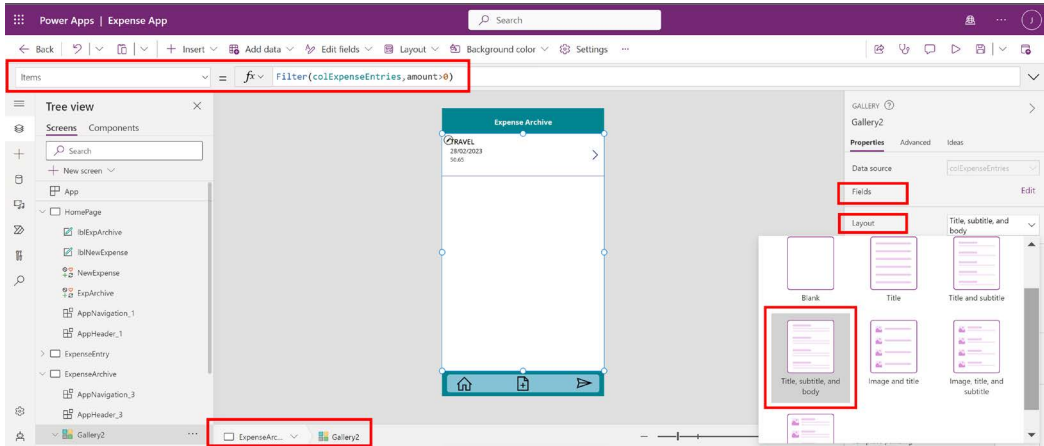


Figure 16.49: Editing gallery fields

9. Move back to the **HomePage** screen and select the **ExpArchive** button so that the **OnSelect** property can be set to navigate a user to the newly created **ExpenseArchive** screen:

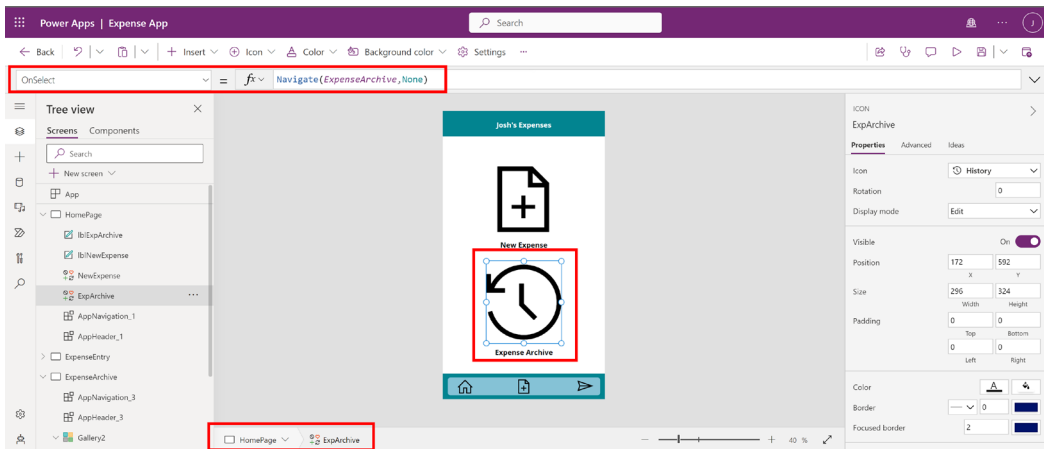


Figure 16.50: Adding logic to the ExpArchive button

10. Return to the **Components** area of the **Tree view** and add a new custom property for **AppNavigation** so that a user can navigate to the Expense Archive page:

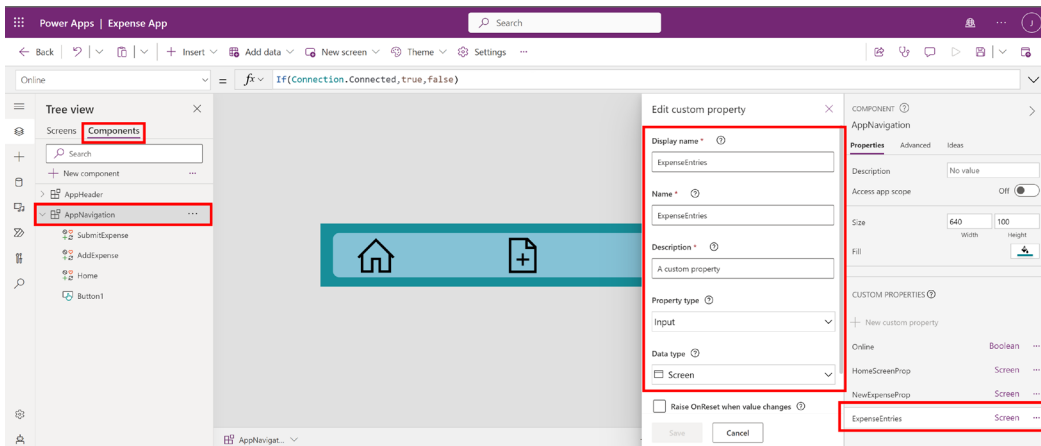


Figure 16.51: Adding AppNavigation properties

Alter the **OnSelect** property of the **SubmitExpense** button within the **AppNavigation** component. Ensure that the property is updated on each screen's **AppNavigation** component so that a user can move to the intended screen no matter where they are in the app:

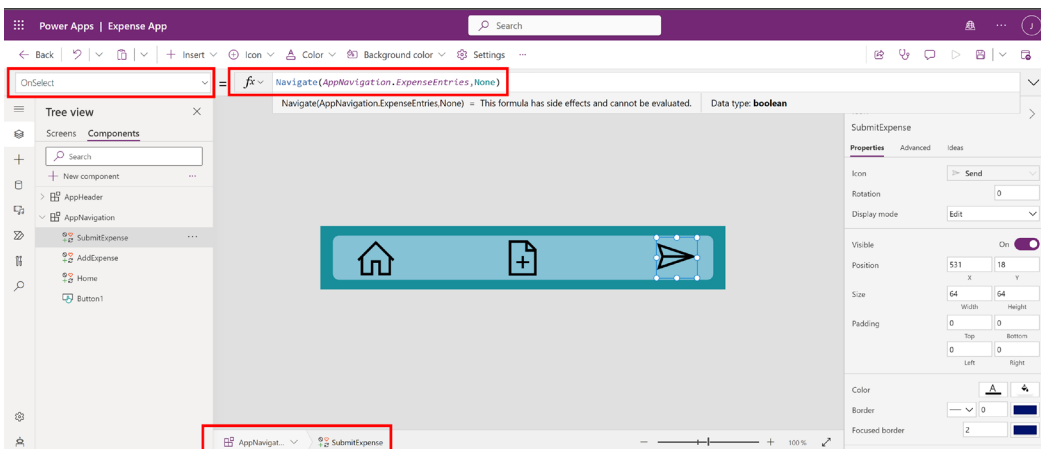


Figure 16.52: Altering SubmitExpense button properties

11. Add a new button to the **ExpenseArchive** screen. Using the provided Power Fx formula, add the code to the **OnSelect** property of the newly created button. Rename the button **Submit**:

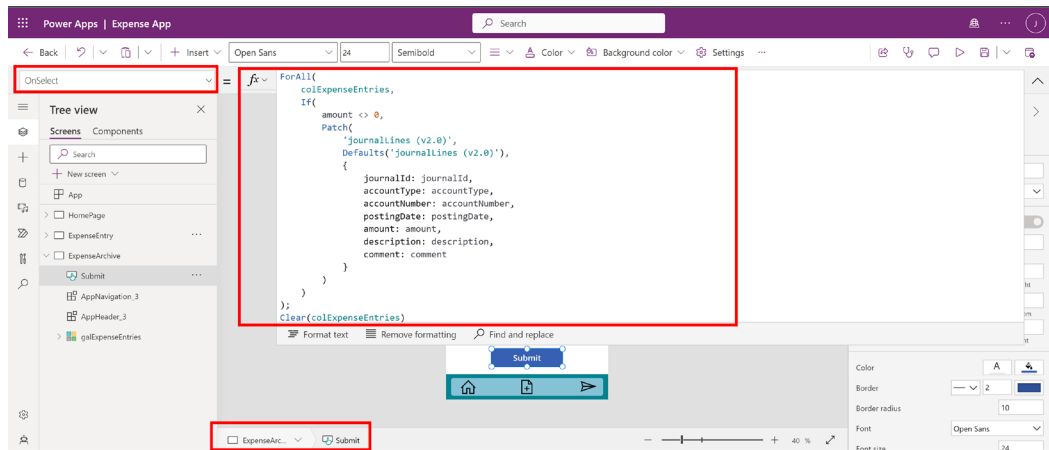


Figure 16.53: Adding OnSelect code to the SubmitExpense button

Here's the Power Fx formula to insert for the OnSelect property:

```
ForAll(
    colExpenseEntries,
    If(
        amount <> 0,
        Patch(
            'journalLines (v2.0)',
            Defaults('journalLines (v2.0)'),
            {
                journalId: journalId,
                accountType: accountType,
                accountNumber: accountNumber,
                postingDate: postingDate,
                amount: amount,
                description: description,
                comment: comment
            }
        )
    )
);
Clear(colExpenseEntries)
```

12. Change the **Visible** property of the **Submit** button so that only a user who is online can use the button to submit the expense entries to Business Central. Note that a further condition has been added so that only when the collection for the expense entries is more than or equal to one it displays the button:

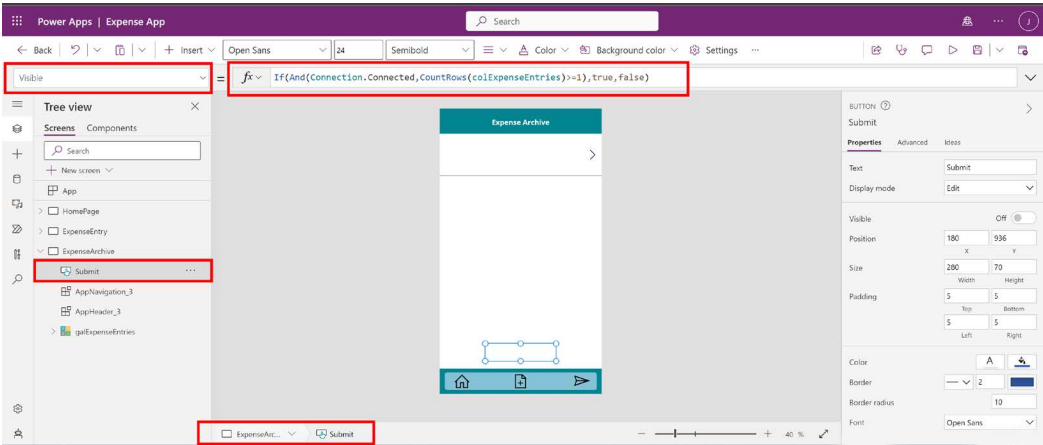


Figure 16.54: Controlling the display of the Submit button

13. When **Submit** is used, it pushes data into Business Central as we intended. Note that entries with the same **Document No.** are pairs. However, the dimension that was set up is not populated. This is because the Business Central API page, which has been used to push the data from the Power Apps app to Business Central, does not have this field:

Posting Date	Document No.	Account Type	Account No.	Account Name	Description	Amount (LCY)	Expense-type Code	Comment
28/02/2023	G00013	G/L Account	31550	Employee Expenses	Train	50.65		TRAVEL
28/02/2023	G00013	Employee	E0010	Josh Anglesea	Train	-50.65		TRAVEL
28/02/2023	G00014	G/L Account	31550	Employee Expenses	Train	50.65		TRAVEL
28/02/2023	G00014	Employee	E0010	Josh Anglesea	Train	-50.65		TRAVEL
28/02/2023	G00015	G/L Account	31550	Employee Expenses	Train	50.65		TRAVEL
28/02/2023	G00015	Employee	E0010	Josh Anglesea	Train	-50.65		TRAVEL

Figure 16.55: Data pushed by Submit

14. This can be remedied if we alter the point that an entry is submitted. To do this, the **SaveExpense** button will need altering. The **OnSelect** property Power Fx formula will be adjusted, and so will the **Text** property.

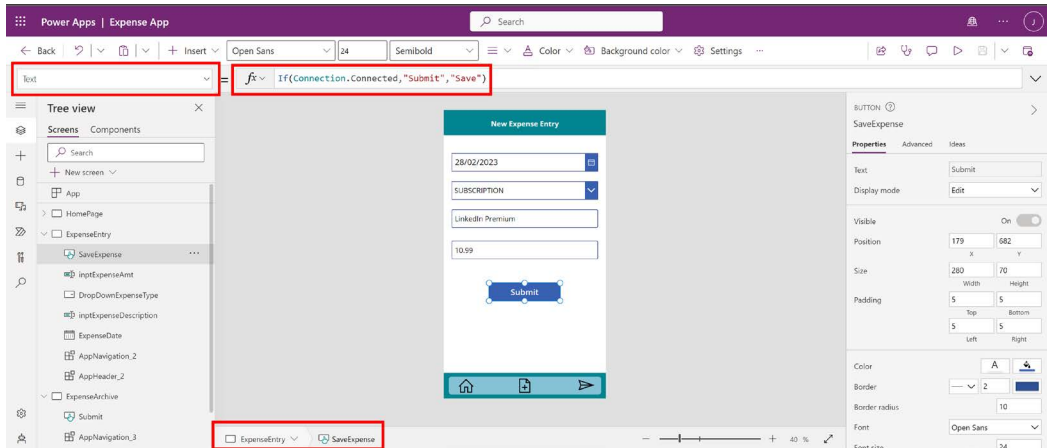


Figure 16.56: Altering the OnSelect property of SaveExpense

Power Fx formula for OnSelect:

```

If(
    Connection.Connected,
    //The variable of varJournalLineID stores the response from the Patch()
    //function. This will have the journalLineId which is required to add a
    //dimension to a journal line through the API page
    Set(
        varJournalLineID,
        Patch(
            'journalLines (v2.0)',
            Defaults('journalLines (v2.0)'),
            {
                journalId: varJournalID,
                accountType: "G/L Account",
                accountNumber: "31550",
                postingDate: Today(),
                amount: Value(inptExpenseAmt.Text),
                description: inptExpenseDescription.Text,
                comment: DropDownExpenseType.SelectedText.code
            }
        )
    );

```

```

Patch(
    'journalLines (v2.0)',
    Defaults('journalLines (v2.0)'),
    {
        journalId: varJournalID,
        accountType: "Employee",
        accountNumber: First(colEmployees).number,
        postingDate: Today(),
        amount: Value(inptExpenseAmt.Text),
        description: inptExpenseDescription.Text,
        comment: DropDownExpenseType.SelectedText.code
    }
),
//Expense Entry
Collect(
    colExpenseEntries,
    {
        journalId: varJournalID,
        accountType: "G/L Account",
        accountNumber: "31550",
        postingDate: ExpenseDate.SelectedDate,
        amount: Value(inptExpenseAmt.Text),
        description: inptExpenseDescription.Text,
        comment: DropDownExpenseType.SelectedText.code
    }
);

//Balancing Entry
Collect(
    colExpenseEntries,
    {
        journalId: varJournalID,
        accountType: "Employee",
        accountNumber: First(colEmployees).number,
        postingDate: ExpenseDate.SelectedDate,
        amount: Value(inptExpenseAmt.Text) - Value(inptExpenseAmt.
Text)*2,
        description: inptExpenseDescription.Text,
        comment: DropDownExpenseType.SelectedText.code
    }
);

```

```

SaveData(
    colExpenseEntries,
    "colExpenseEntries"
)
);
Reset(ExpenseDate);
Reset(inptExpenseAmt);
Reset(inptExpenseDescription);
Reset(DropDownExpenseType)

```

Now that the Power Fx formula for the SaveExpense button's OnSelect property has been altered, a new function can be added, which will take care of assigning a dimension value for one of the general journal lines (only the G/L account line requires this).

## Part 6: Calling a Power Automate flow to execute actions in the app

The new function will come from a Power Automate flow, which can be called directly by Power Apps with the data we pass to it. To add a flow, follow these steps:

1. From the Power Automate Tree view, select **Create new flow**:

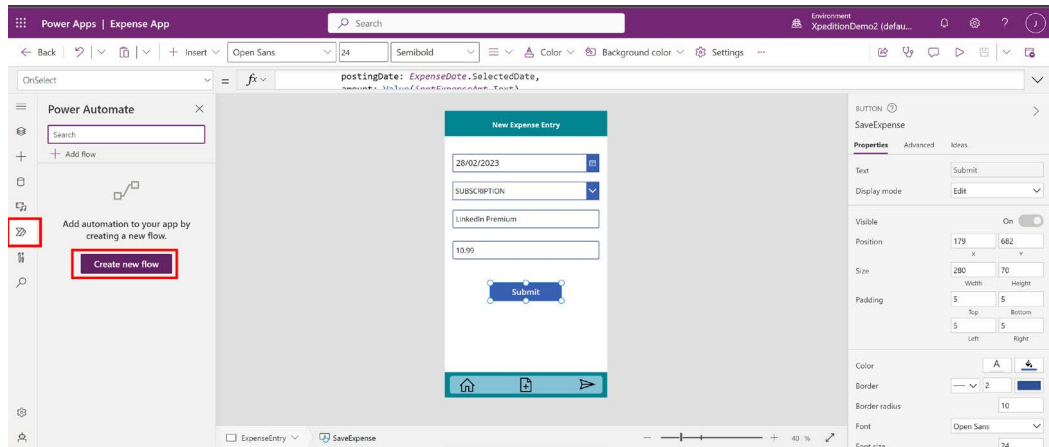


Figure 16.57: Power Automate in the Tree view

- This will open the **Create your flow** screen. From here you can either choose to create a flow from blank, or add a flow following a template. In our case, we will select **Create from blank**:

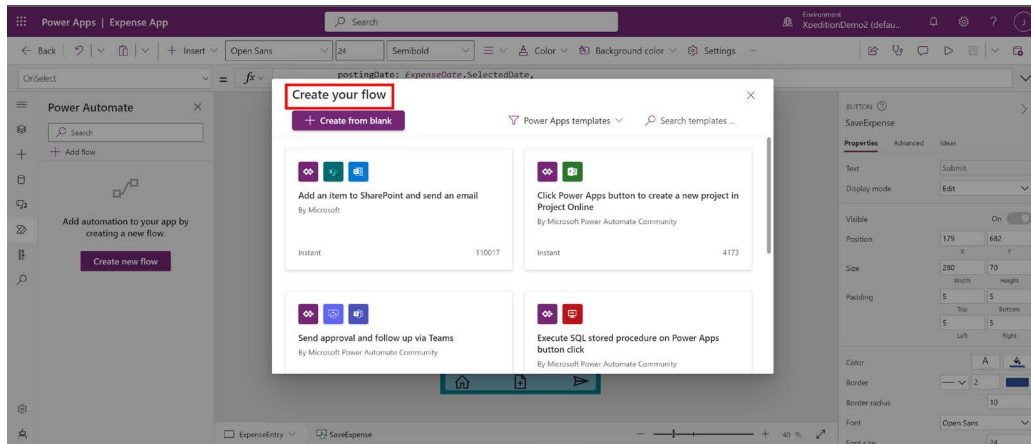


Figure 16.58: Create your flow screen

- The **Create your flow** screen will then open a flow builder. Give the flow an appropriate name. Note that the flow will start with a Power Apps trigger, after which we will add in the other steps:

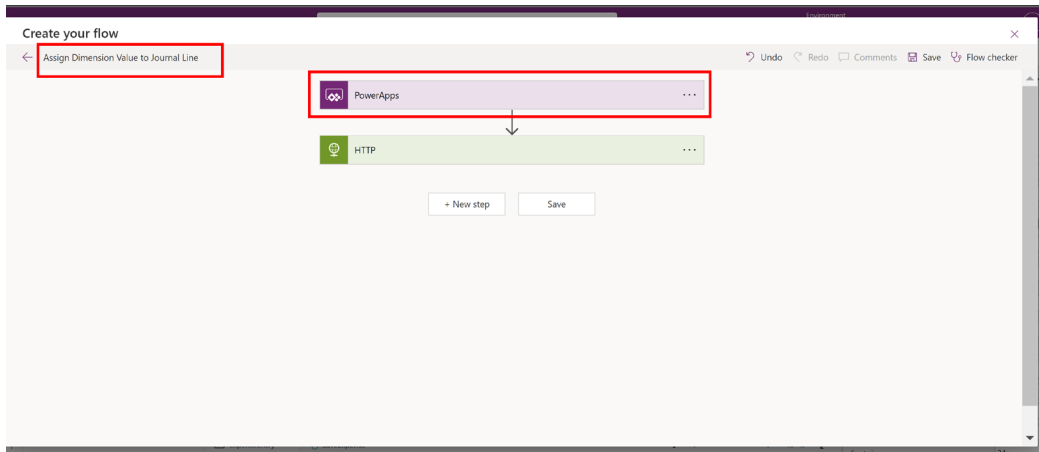


Figure 16.59: Adding a flow trigger

- As you can see, we need to set up an HTTP endpoint.

The HTTP structure is bound to the journal line, in our case. The required call will be structured as follows:

```
POST businesscentralPrefix/companies({id})/journalLines({id})/
dimensionSetLines
```

The endpoint for Business Central dimensionSetLines is explained here: [https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/api-reference/v2.0/api/dynamics\\_dimensionsetline\\_create](https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/api-reference/v2.0/api/dynamics_dimensionsetline_create).

- With the HTTP endpoint added to the flow, we will now need to set up an HTTP action. This is needed because the Business Central Power Automate connector is unable to handle this type of request. In our case, a cloud instance of Business Central will be used and therefore OAuth 2.0 is utilized with API calls.

This blog outlines how to use the Power Automate HTTP connector with OAuth 2.0 for Business Central: <https://joshanglesea.wordpress.com/2022/03/01/business-central-oauth2-for-power-automate-%f0%9f%9b%82/>. Alternatively, this is another option which yields the same result: <https://joshanglesea.wordpress.com/2023/03/14/business-central-custom-connector-for-standard-api/>.

Once you have finished reading the blog, you can add the required HTTP action to assign a dimension value to a journal line.

- Now that we have an HTTP action to assign a value to a journal line, we need to pass data between the Power Apps app and our flow. To do so, we will use the **Ask in PowerApps** dynamic content. The following screenshot shows the use of dynamic content (1), the creation of a variable (2), and the ability to apply the content to other sections (3):

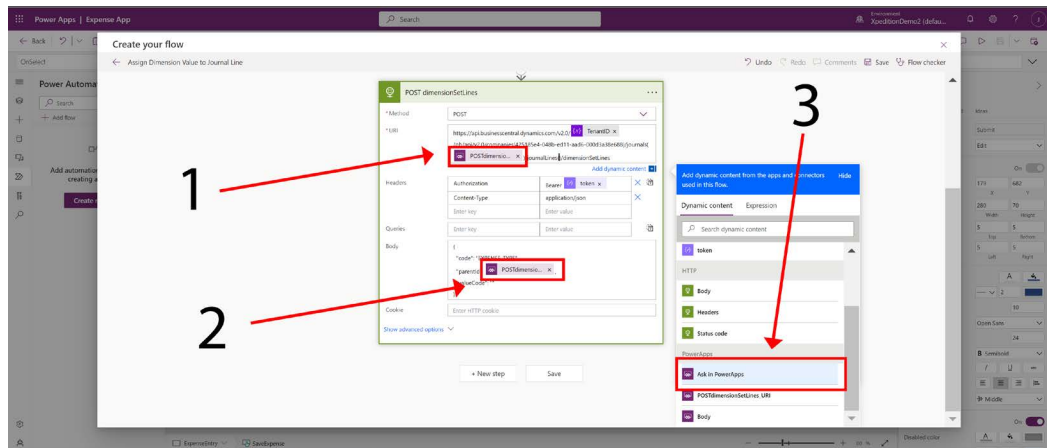


Figure 16.60: Adding dynamic content to a flow

The URI being used for this dynamic content, in order to pass the journal line Power Apps data, is the following:

```
https://api.businesscentral.dynamics.com/v2.0/<Your Tenant ID>/gb/api/v2.0/companies(425185e4-048b-ed11-aad6-000d3a38e688)/journals(<Ask in PowerApps Variable 1>)/journalLines(<Ask in PowerApps Variable 2>)/dimensionSetLines
```

The body is a JSON object. It should look like this:

```
{
  "code": "EXPENSE-TYPE",
  "parentId": "<Ask in PowerApps Variable 1>",
  "valueCode": "<Ask in PowerApps Variable 3>"
}
```

7. With the dynamic content configured, you now have a flow that assigns a dimension value for one of the journal lines. Great! Once the flow has been saved, you will be returned to make.powerapps.com and the flow will be available for access directly in the Power Apps app:

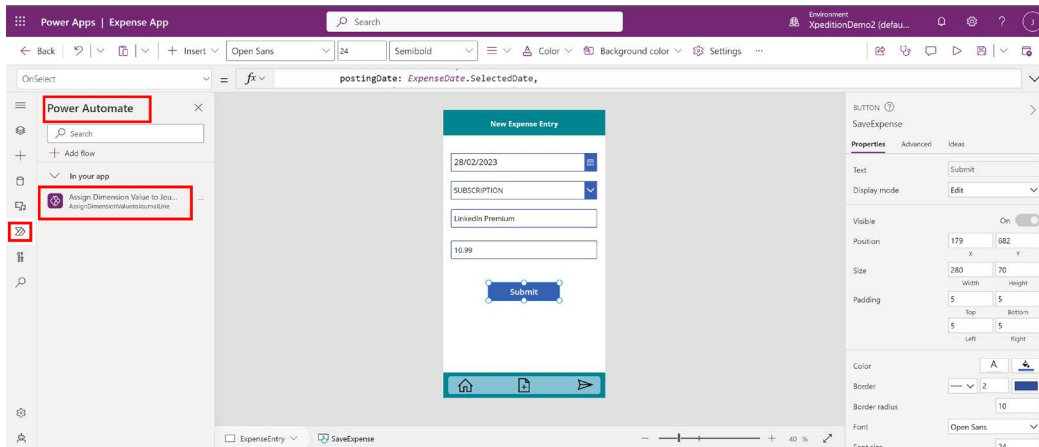


Figure 16.61: Flow shown under Power Automate in the Tree view

8. The flow can then be called in the OnSelect property of the **SaveExpense** button. The flow is called online if the app is online.

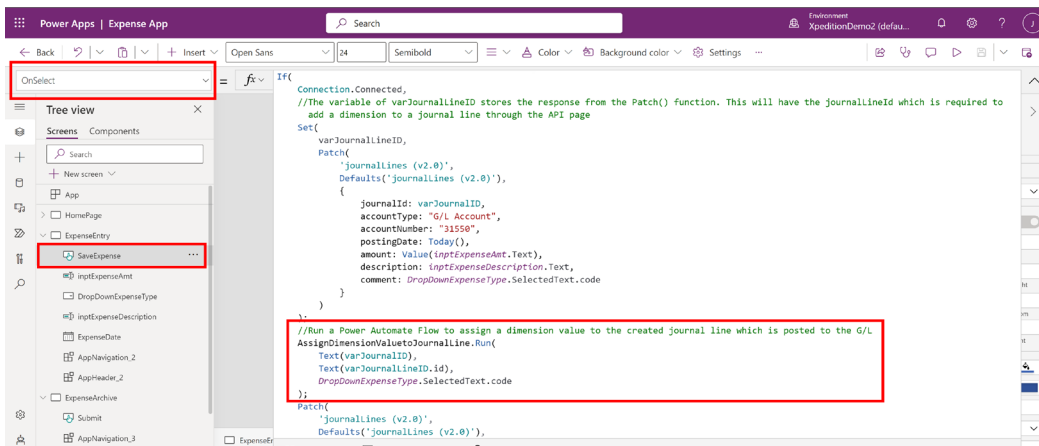
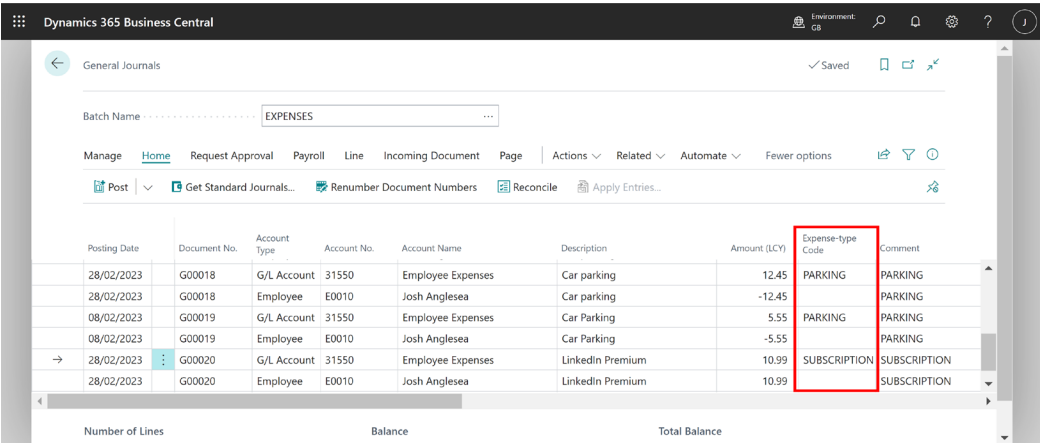


Figure 16.62: Calling the flow in OnSelect

When the **SaveExpense** button's **OnSelect** logic is executed, the end result in Dynamics 365 Business Central is as shown in the following:



The screenshot shows the Dynamics 365 Business Central interface for General Journals. The batch name is 'EXPENSES'. The table below shows the resulting journal entries:

Posting Date	Document No.	Account Type	Account No.	Account Name	Description	Amount (LCY)	Expense-type Code	Comment
28/02/2023	G00018	G/L Account	31550	Employee Expenses	Car parking	12.45	PARKING	PARKING
28/02/2023	G00018	Employee	E0010	Josh Anglesea	Car parking	-12.45	PARKING	PARKING
08/02/2023	G00019	G/L Account	31550	Employee Expenses	Car Parking	5.55	PARKING	PARKING
08/02/2023	G00019	Employee	E0010	Josh Anglesea	Car Parking	-5.55	PARKING	PARKING
28/02/2023	G00020	G/L Account	31550	Employee Expenses	LinkedIn Premium	10.99	SUBSCRIPTION	SUBSCRIPTION
28/02/2023	G00020	Employee	E0010	Josh Anglesea	LinkedIn Premium	10.99	SUBSCRIPTION	SUBSCRIPTION

Figure 16.63: SaveExpense logic results

In this section, you learned how to create a real-world complex Power Apps application connected to Dynamics 365 Business Central, complete with custom logic and offline capabilities.

In the next section, we'll see how to expose Dynamics 365 Business Central data to Dataverse using Dataverse virtual tables.

# Exposing Dynamics 365 Business Central data to Dataverse by using virtual tables

Dataverse is a core part of Microsoft Power Platform. Exposing Dynamics 365 Business Central data to Dataverse opens a wide range of possibilities for integrations.

When talking about integrations between Dynamics 365 Business Central and Dataverse, we mainly have the following options:

- Data sync (replication) between Dynamics 365 Business Central and Dataverse. This is a data replication between the two systems that we don't recommend anymore (and it will not be covered in this book).
- Virtual tables in Dataverse (they enable the integration of data residing in external systems by seamlessly representing that data as tables in Dataverse).
- Data change events: the Dataverse connector supports triggering a workflow if a data change from a Business Central virtual table occurs.
- Business events (new type of events that you can expose from Dynamics 365 Business Central in order to notify an external system).

In this section, we'll talk about **Dataverse virtual tables**.

A virtual table in Dataverse is essentially a table without the associated physical records created in the Dataverse database. During runtime, when a record is required, its state is dynamically retrieved from the associated external system. With virtual tables, data is not replicated to Dataverse but instead remains in the original system (this permits you to save storage space in the Dataverse environment). Virtual tables are enabled for Dynamics 365 Business Central standard and custom APIs and they support **Create**, **Read**, **Update**, and **Delete** operations (CRUD) and also data change events.

To start integrating Dynamics 365 Business Central and Dataverse, you need to execute the **Set up a connection to Dataverse** Assisted Setup wizard from Business Central:

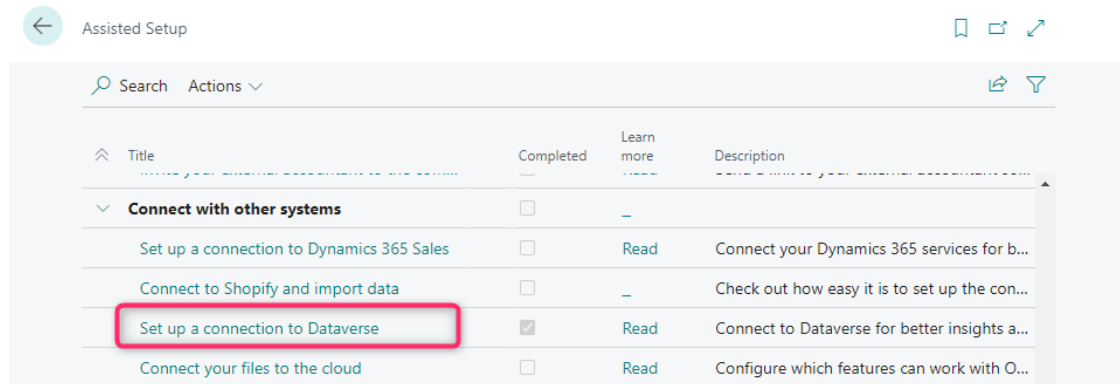



Figure 16.64: Setup wizard for connecting to Dataverse

To use Dataverse virtual tables and business events, you need to enable the **Enable virtual tables and events** option in the **Dataverse Connection Setup** page of the wizard:

### Dataverse Connection Setup



Quickly set up the connection, couple records, and even synchronize data.

Enable data synchronization ..... ☒

Connect Business Central to Dataverse to synchronize data with other business apps.

Quickly set up Business Central virtual tables in Dataverse and enable business events that Business Central sends to Dataverse.

Enable virtual tables and events ... ☒

If you choose Next we will try to find your Dataverse environments so you can choose the one to connect to.

Back Next Finish

*Figure 16.65: Enabling Dataverse virtual tables and business events*

Then, you need to select the Dataverse environment to connect to from the list of the available environments for the user:

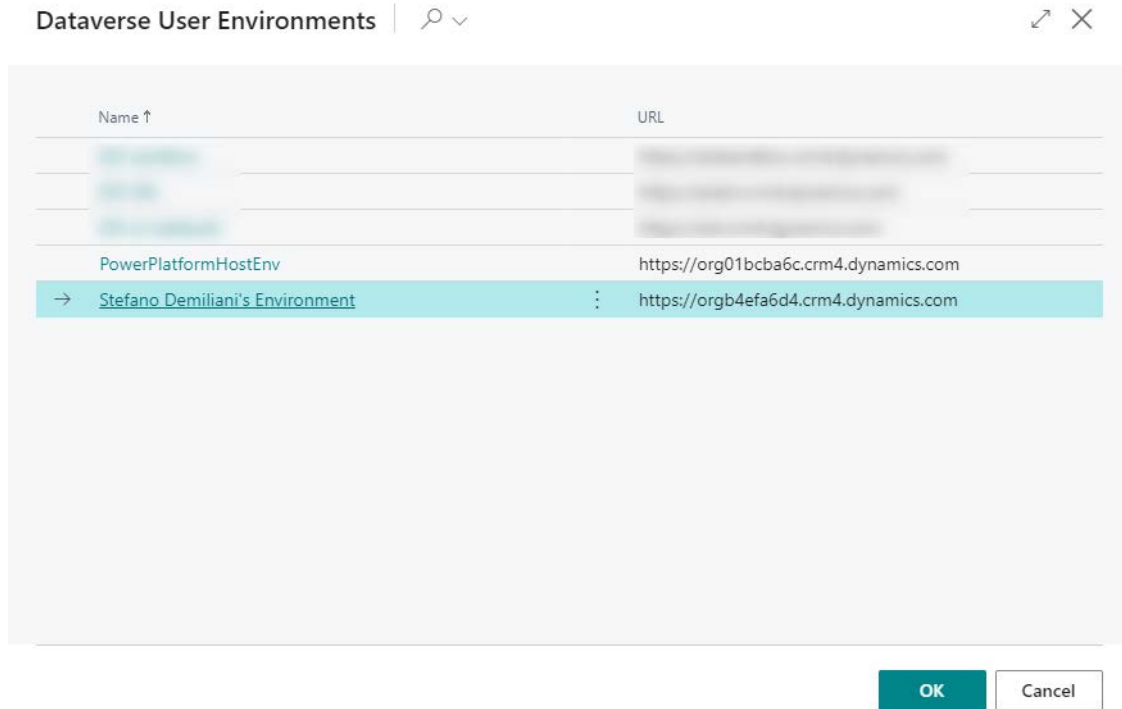





Figure 16.66: Selecting a Dataverse environment

Then, you need to sign in with an administrator user account and give consent to the application that will be used to connect to Dataverse. The account will be used once to install and configure the components that the integration requires.

Then, select **Team** as the ownership model for Dataverse records:

## Dataverse Connection Setup






**Choose an ownership model.**

People or a team own records in Dataverse that are created from data in Business Central. We recommend the Team model.

Team



We will create a business unit and a team in Dataverse. Members of the team will own the synchronized data and can assign records to other users or teams in the business unit.

**Complete setup without synchronization**

Choose this option to enable the connection without synchronizing data.

☒

Back

Next

Finish

Figure 16.67: Selecting an ownership model

At a certain point of the Dataverse connection setup, the wizard asks you to set up virtual tables. From here, you can directly install the **Business Central Virtual Table app** from the Microsoft AppSource marketplace in your Dataverse environment. This app is mandatory in order to be able to use virtual tables.

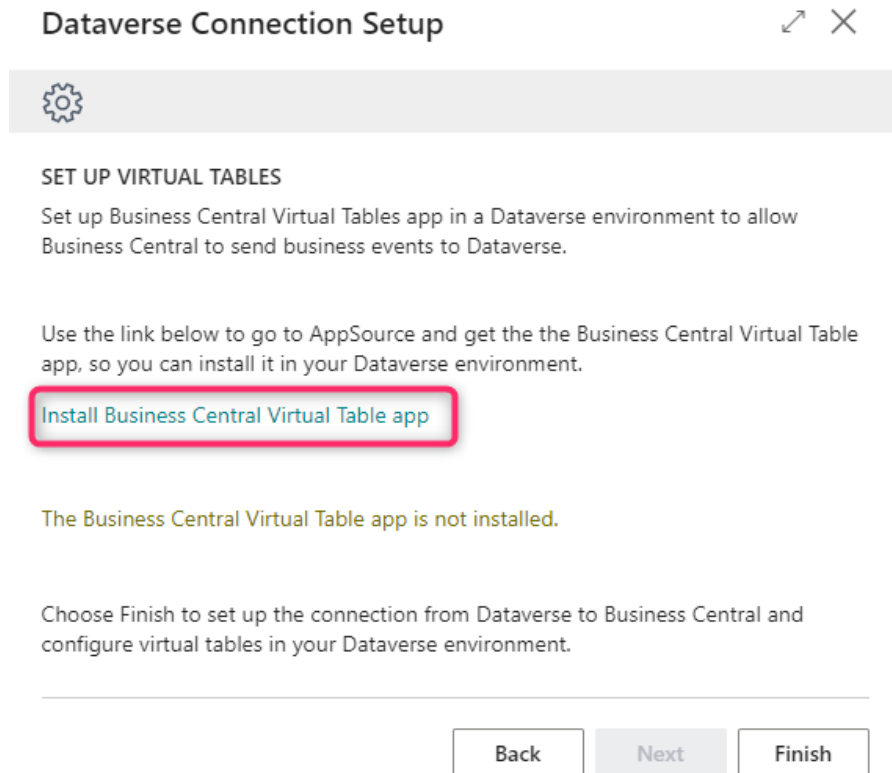



Figure 16.68: Installing the Virtual Table app

By selecting the link in the wizard, you will be redirected to Microsoft's AppSource **Business Central Virtual Table** app page where you can select to install the app. Then, *Power Platform Admin Center* opens and from here you can select the Dataverse environment where the app will be installed:

### Install Business Central Virtual Table (Preview) ×



**Name**  
Business Central Virtual Table (Preview)

**Description**  
Microsoft Dynamics 365 Business Central Virtual Table is a virtual data source in Microsoft Dataverse allowing Create, Read, Update and Delete operations from Microsoft Dataverse against Dynamics 365 Business Central. Data for virtual tables do not reside...  
[See more](#)

**Publisher**  
Microsoft

**Select an environment \***  

Stefano Demiliani's Environment ▼

[Don't see your environment?](#)

☒ I agree to Microsoft's [Legal Terms](#) and [Privacy Statement](#)

☒ I agree to [Privacy Statement](#) and [Legal Terms](#) for importing packages into Dynamics 365

Install

Cancel

Figure 16.69: Selecting an environment for the Virtual Table app

Then, click on Install and the **Business Central Virtual Entity** app will be installed in your selected Dataverse environment.

Together with this app, the following solutions will also be installed in your Dataverse environment:

- **Dynamics365Company:** This adds the `cdm_company` table, which is referenced by all Business Central virtual tables. All communication to Business Central requires the company ID in the request.
- **MicrosoftBusinessCentralVESupport:** This provides the core support for the Business Central virtual table feature.
- **MicrosoftBusinessCentralERPCatalog:** This provides a list of available tables (standard and custom APIs), in a Business Central instance.
- **MicrosoftBusinessCentralVEAnchor:** This acts as a container, holding information needed for AppSource.
- **MicrosoftBusinessCentralERPVE:** This is the solution that contains virtual tables generated for Business Central. Tables are added automatically at runtime once they're made visible in the `MicrosoftBusinessCentralERPCatalog`.

When the app is installed, you can start using Dynamics 365 Business Central virtual tables on Dataverse. A user named Dynamics 365 Business Central for Virtual Tables will be created in Business Central to handle the communication.

It's important to note that, by default, the entities exposed by Dynamics 365 Business Central as APIs (standard or custom) are not available as Dataverse virtual tables. If you want to expose a Dynamics 365 Business Central entity as a virtual table in Dataverse, you need to enable it.

To do that, from your Dataverse environment, select **Tables** and then search for **Available Business Central table** (by making sure to search for **All** and not just for **Recommended** tables).

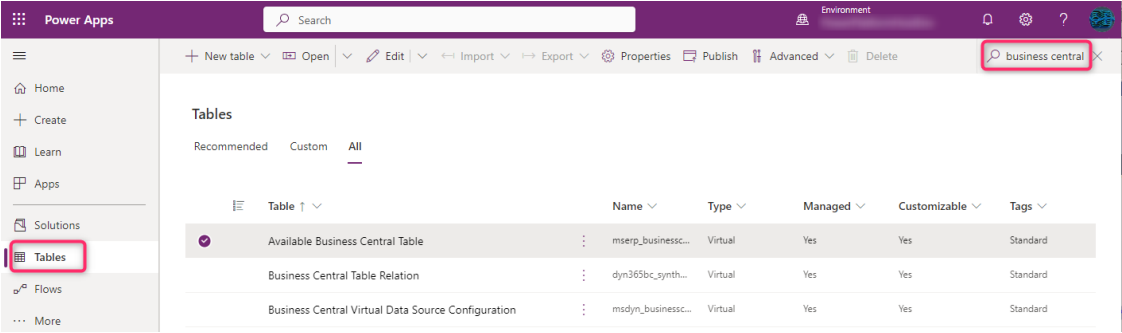


Figure 16.70: Searching for Available Business Central Table

From here, you can see the list of available tables (APIs exposed from Dynamics 365 Business Central) that you can enable. To enable an entity as a virtual table, set the **Visible** field to **Yes** and save:

Available Business Central...

Name * ↑	Display Name	Visible	API Route
balanceSheet	Balance Sheet	No	v2.0
bankAccount	Bank Account	No	v2.0
cashFlowStatement	Cash Flow Statement	No	v2.0
companyInformation	Company Information	No	v2.0
contact	Contact	No	v2.0
contactInformation	Contact Information	No	v2.0
countryRegion	Countries Region	No	v2.0
currency	Currency	No	v2.0
✓ customer	Customer	Yes	v2.0
customerFinancialDetail	Customer Financial Detail	No	v2.0
customerPayment	Customer Payment	No	v2.0
customerPaymentJournal	Customer Payment Journal	No	v2.0
customerReturnReason	Customer Return Reason	No	v2.0

Figure 16.71: Setting the Visible field for an available table

This will generate the virtual table in the MicrosoftBusinessCentralERPVE solution, which you can start using from any Power Platform solution exactly like any native Dataverse table.

In the next section, we'll learn how to create events in AL code and expose them to Dataverse by using business events.

## Exposing Dynamics 365 Business Central events to Dataverse

**Business events** are a way of notifying and triggering an external system (mainly Dataverse but not only) when an action occurs in Dynamics 365 Business Central business logic.

The following diagram shows how **Business events** relates to Dataverse integration:

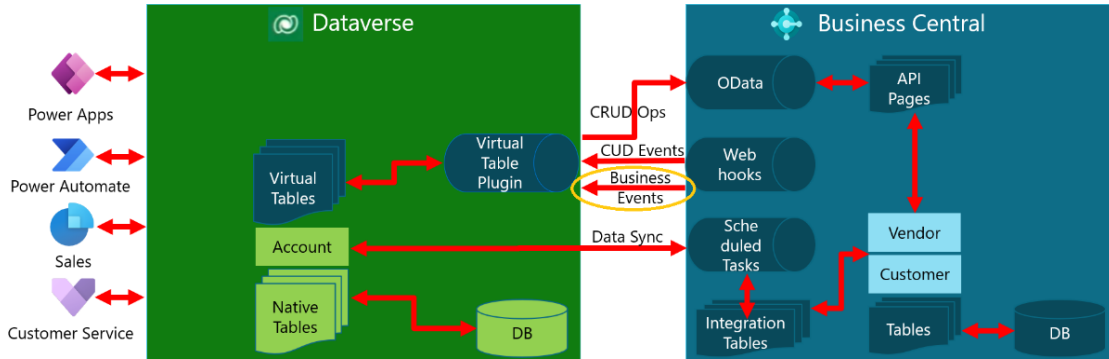


Figure 16.72: Business events diagram

They are published by Dynamics 365 Business Central using webhook techniques and they are exposed to Dataverse via the virtual table plugin. In Dataverse processes, you can react to those events.

To use Business events from Dynamics 365 Business Central, you need to execute the **Dataverse Connection Setup** (as previously explained) and from here select the **Enable virtual tables and events** flag.

A business event can be defined via AL code in Dynamics 365 Business Central by using the `ExternalBusinessEvent` attribute:

```
[ExternalBusinessEvent(Name: Text, DisplayName: Text, Description: Text,
Category: enum)]
```

As an example, imagine that we want to create a business event to notify an external system when a sales order in Dynamics 365 Business Central is posted.

In our AL code, we can create an event subscriber to a standard Business Central event (`OnPostDocumentBeforeNavigateAfterPosting` of the **Sales Order** page), and from this event subscriber, we can raise our external business event:

```
[EventSubscriber(ObjectType::Page, Page::"Sales Order",
OnPostDocumentBeforeNavigateAfterPosting, '', false, false)]
    local procedure OnPostDocument(var SalesHeader: Record "Sales Header";
var PostingCodeunitID: Integer; var Navigate: Enum "Navigate After Posting";
DocumentIsPosted: Boolean; var IsHandled: Boolean)
    begin
        SalesOrderPosted(SalesHeader.SystemId, SalesHeader."Sell-to Customer
Name", SalesHeader."No.");
    end;
```

The SalesOrderPosted business event can be defined as follows:

```
[ExternalBusinessEvent('SalesOrderPosted', 'Sales Order Posted', 'Triggered
when a Sales Order is posted in BC', EventCategory::Sales)]
    [RequiredPermissions(PermissionObjectType::TableData, Database::"Sales
Header", 'R')] //optional
    procedure SalesOrderPosted(salesOrderId: Guid; customerName: Text[250];
orderNo: Text[250])
    begin

    end;
```

As per the definition, we have also added a category for our external business events (called Sales). To do that, you can create an enumextension object of the standard EventCategory enum object as follows:

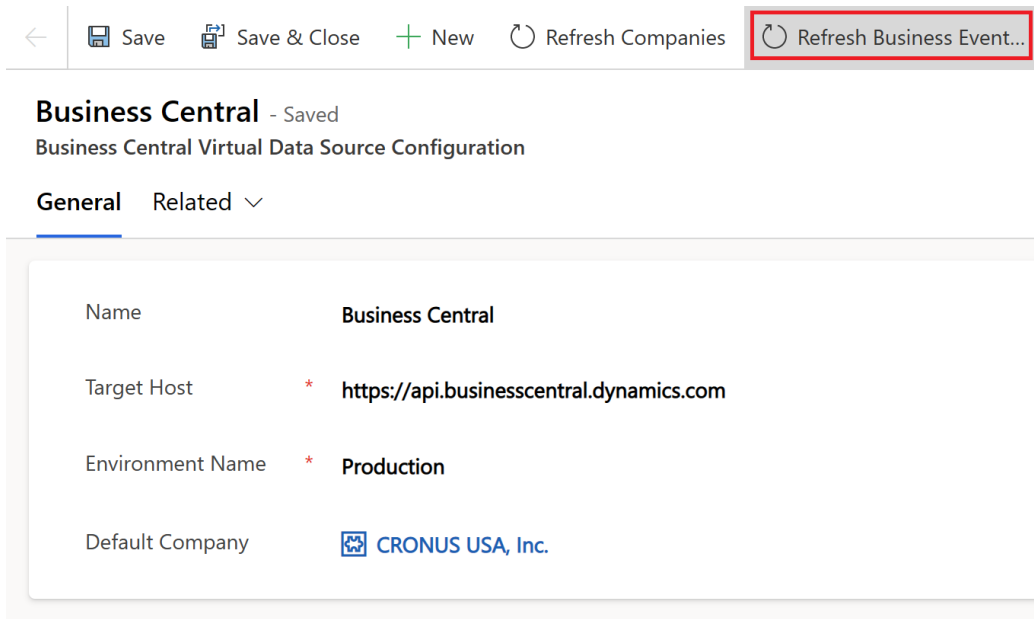
```
enumextension 50100 SDEventCategoryExt extends EventCategory
{
    value(50100; Sales)
    {
        Caption = 'Sales';
    }
}
```

When you publish an AL extension containing a business event definition, the events are available to external systems.

To refresh the business event catalog in Dataverse after installing your AL extension, you need to sign in to the Power Apps portal ([make.powerapps.com](https://make.powerapps.com)) and select your Dataverse environment. Here:

1. Select **Tables**, then search for **Business Central Virtual Data Source Configuration** under the **All** tab, and then select it.
2. Select **Edit**, then the **Business Central** row, and then select **Edit row using form** to open a form.

3. Select **Refresh Business Event Catalog** on the form:

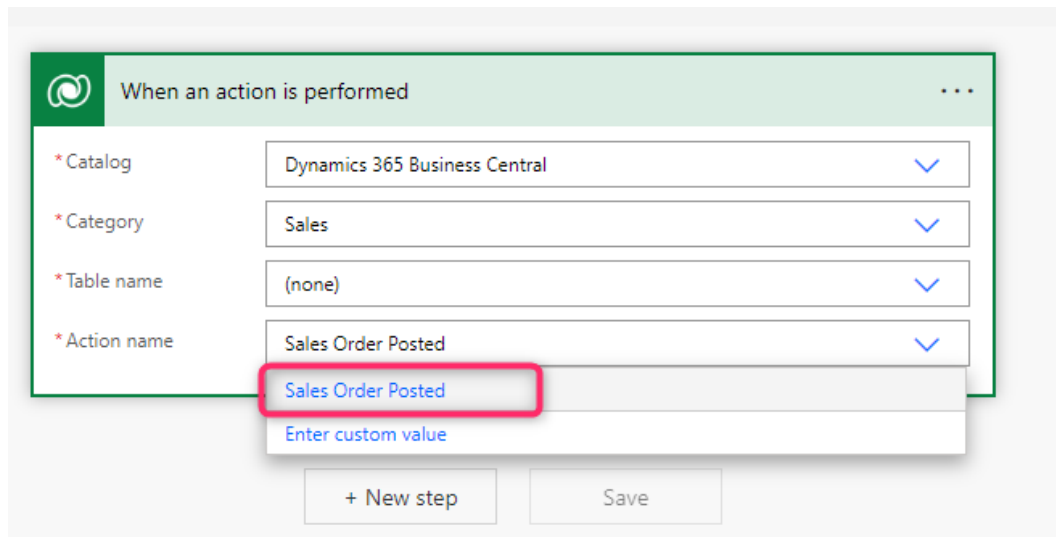


The screenshot shows the 'Business Central' configuration form. At the top, there is a toolbar with buttons: 'Save', 'Save & Close', 'New', 'Refresh Companies', and 'Refresh Business Event...'. The 'Refresh Business Event...' button is highlighted with a red rectangular box. Below the toolbar, the form title is 'Business Central - Saved', followed by 'Business Central Virtual Data Source Configuration'. There are two tabs: 'General' (selected) and 'Related'. The 'General' tab contains a table with the following fields:

Name	Business Central
Target Host	* https://api.businesscentral.dynamics.com
Environment Name	* Production
Default Company	CRONUS USA, Inc.

Figure 16.73: Refreshing the event catalog

Now, the business events are visible to Dataverse. To use them, you can create a new Power Automate flow, select the **Dataverse connector**, and use the **When an action is performed** trigger:



The screenshot shows the 'When an action is performed' trigger configuration in Power Automate. The trigger is set to 'Dynamics 365 Business Central'. The configuration fields are:

- \* Catalog: Dynamics 365 Business Central
- \* Category: Sales
- \* Table name: (none)
- \* Action name: Sales Order Posted

The 'Sales Order Posted' action name is highlighted with a red rectangular box. Below the configuration fields, there are two buttons: '+ New step' and 'Save'.

Figure 16.74: Using business events in Power Automate

Here, you need to:

1. Set Dynamics 365 Business Central as **Catalog**.
2. Set Sales as **Category** (our newly created event category).
3. Set **(none)** as **Table name**.
4. Select your external business event in the **Action name** field

Then, you can add the desired actions to your workflow. The workflow will be triggered when the external business events are raised from the AL code in Dynamics 365 Business Central.

External business events can also be used directly from the Dynamics 365 Business Central connector. Here is an example of a workflow triggered by a business event using the standard connector:

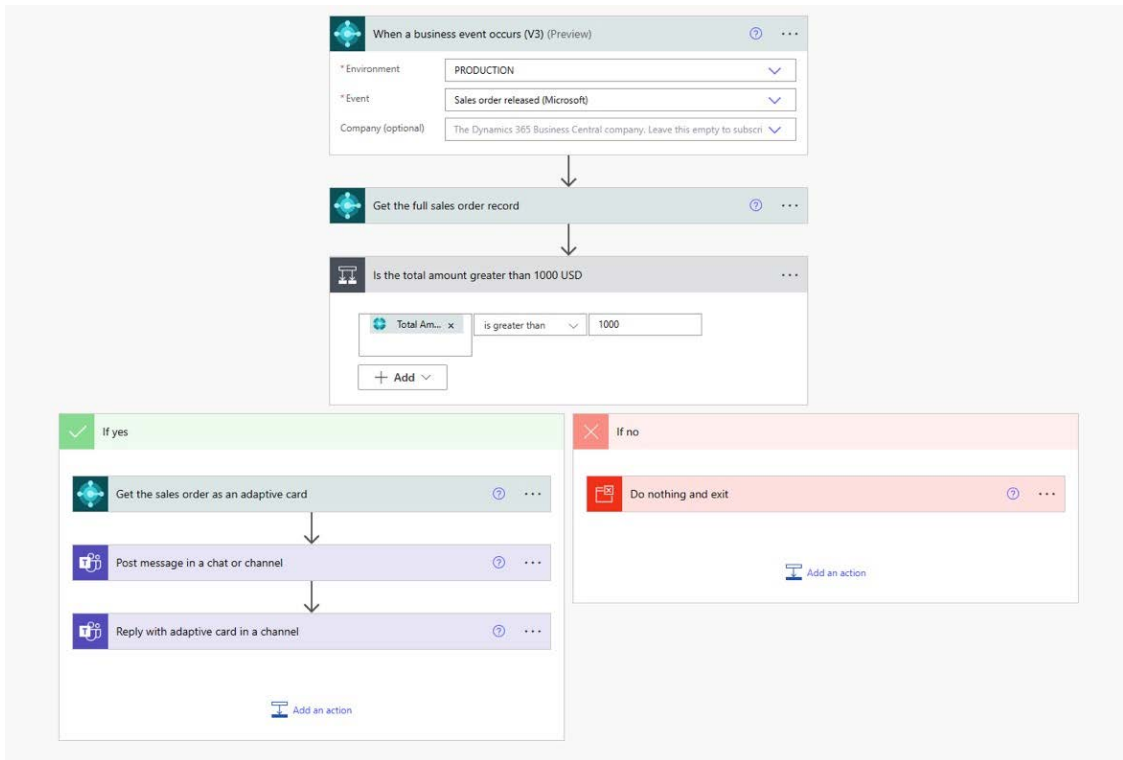


Figure 16.75: Using business events alongside the Business Central connector

## Summary

In this chapter, we learned about the various ways to integrate Dynamics 365 Business Central with Microsoft Power Platform. We learned how to use Power Automate with Dynamics 365 Business Central, how to create Power Apps working with Dynamics 365 Business Central data, how to expose Dynamics 365 Business Central data to Dataverse by using Dataverse virtual tables, and how to create business events from AL code that can be consumed from Dataverse.

This set of features gives you the full possibility of integrating Dynamics 365 Business Central with every Dynamics 365 and Power Platform application.

In the next chapter, we'll give an overview of useful Visual Code extensions for developing solutions with Dynamics 365 Business Central that can help you in various tasks during the development phase.

## Leave a review!

*Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.*



*\*Limited Offer*

# 17

## Useful and Proficient Tools for AL Developers

In previous chapters, we provided some guidance and best practices for developing simple and more exotic solutions with Dynamics 365 Business Central.

Having the correct tools when working with extensions and Visual Studio Code can save you lots of time and energy. In this chapter, we want to give you an overview of some third-party development tools that you can use in your everyday developer life with AL to be more proficient in many tasks. We will focus on tools developed by a famous name in the Microsoft Business Applications world (Eric Wauters, aka Waldo) and provide an overview of several other interesting and proficient extensions for Visual Studio Code.

While we think many of the points in this chapter might be subjective, it is worth keeping in mind that there could be AL developers who think differently from us, so we want to make it clear that we are personally recommending a tool, rather than saying it will certainly be popular.

We will cover the following topics in this chapter:

- Waldo's tools and how to use them
- A list of several proficient free Visual Studio Code extensions for AL developers

### Who is Waldo?

Waldo's real name is Eric Wauters, and he is one of the founding partners of *iFacto Business Solutions and Cloud Ready Software*. With his 18 years of technical expertise, he is an everyday inspiration to its development teams. As a development manager, he continually acts upon the technical readiness of iFacto.

Apart from that, Eric is also very active in the Microsoft Dynamics 365 Business Central community, where he tries to solve technical issues and shares his knowledge with other Dynamics enthusiasts. Surely, a lot of you will have read some of Eric's posts, which he invariably signs with `waldo`.

Lots of people have been using and even contributing to tools he shares for free on MiBuSo, GitHub, the PowerShell Gallery, and the Visual Studio Marketplace.

His proven track record led to him being given a Microsoft **Most Valuable Professional (MVP)** award each year since 2007.

After learning about Waldo, in the next section, we'll see an overview of most of his tools for AL developers.

## What tools to use

By the time this book was in its final draft, Waldo published a must-view webcast related to this argument. It is worth watching it right after finishing this last chapter: 20231204 - "The" BC Toolkit for developers (<https://www.youtube.com/watch?v=WdD8heDU8es>).



Figure 17.1: Waldo webcast 20231204 - "The" BC Toolkit for developers

Over the years, Waldo has created quite a lot of tools. The first tool Waldo ever put online was back in 2004, WaldoNavPad, a tool that helps to work with bigger texts in Microsoft Dynamics NAV. It helped to break up code into smaller pieces, which was necessary because, back then, we could only have a maximum of 250 characters in one field.

The tool was downloaded over 11,000 times from MiBuSo. Because of its popularity, Waldo updated the tool to a version that worked in the RTC and as an AL extension, where he extended the functionality a bit to have an HTML editor inside Business Central.

Following this tool, quite a few minor tools made it to the download list of MiBuSo, which you can find at: <https://mibuso.com/downloads/results?keywords=waldo>.

Since 2013, when Microsoft released more and more PowerShell building blocks, Waldo decided to dive into that to help the uptake in the community. This resulted in some very extended libraries of helper functions, which are categorized and published on the PowerShell Gallery. Just search for `waldo` (<https://www.powershellgallery.com/packages?q=waldo>).

Every single script in which he uses these modules is online on his GitHub: <https://github.com/waldo1001/Cloud.Ready.Software.PowerShell>. You will find all the modules there and the scripts in which he puts these modules to use.

Many of these PowerShell scripts were created to be able to help the development of V1 extensions. But when these were discontinued, there was a new kid on the block: Visual Studio Code, in which we can develop for modern extensions V2. This tool needed some help with the following:

- Automatically naming files
- Running objects
- Snippets

So, Waldo started to build an extension for Visual Studio Code to help AL developers do their jobs more efficiently. **Waldo's CRS AL Language Extension** was born: <https://marketplace.visualstudio.com/items?itemName=waldo.crs-al-language-extension>.

This is just a peek at Waldo's tools and how he came to build them. In this chapter, we will talk about a few of his tools, focused on making your life as an AL developer a little bit easier.

## The AL Extension Pack

The smallest tool that Waldo ever built is **AL Extension Pack**. In fact, it's a collection of all the **Visual Studio Code** extensions that Waldo values and uses in everyday development tasks.

You can find the **extension pack** on Marketplace with the following link: <https://marketplace.visualstudio.com/items?itemName=waldo.al-extension-pack>.

This is what the home page looks like:

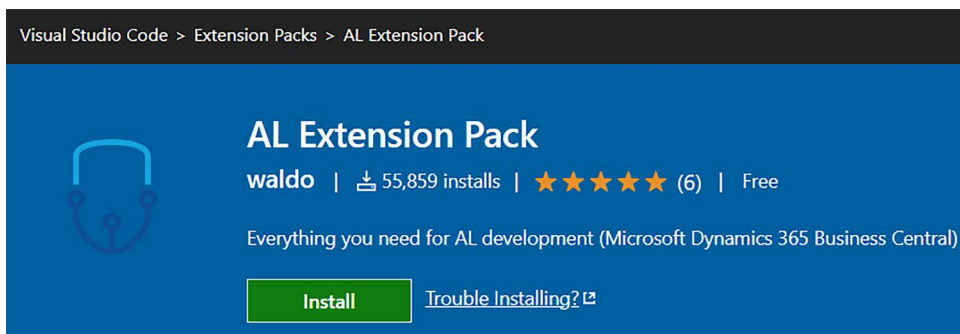


Figure 17.2: AL Extension Pack home page

By simply installing this extension, it will automatically install all extensions that are in the pack, and when Waldo adds an extension, it will automatically be installed on your system as well.

In the previous edition of this book, back in December 2019, there were just 16 extensions listed while today there are 26, meaning that the enrichment of the modern AL development environment is ever growing.

If you do not like or need some of the apps included, you could always disable them and use a Visual Studio Code profile to tailor your development environment at best. You do not need an XLIFF editor when you do not have to manage translations for instance, or you might not like the effect of the bracket colorizer and the TODO highlighter. This is totally up to you.

There is also a similar package that we highly recommend installing if you want to have a full-featured Visual Studio Code environment:

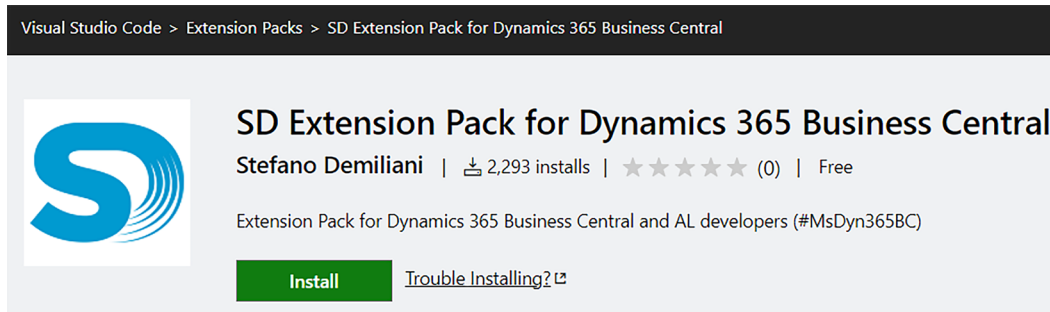


Figure 17.3: SD Extension Pack for Dynamics 365 Business Central

You can find this tool at the following link: <https://marketplace.visualstudio.com/items?itemName=StefanoDemiliani.sd-extpack-d365bc>.

## The waldo's CRS AL Language Extension

A somewhat bigger extension that Waldo has written for the community is **waldo's CRS AL Language Extension**:

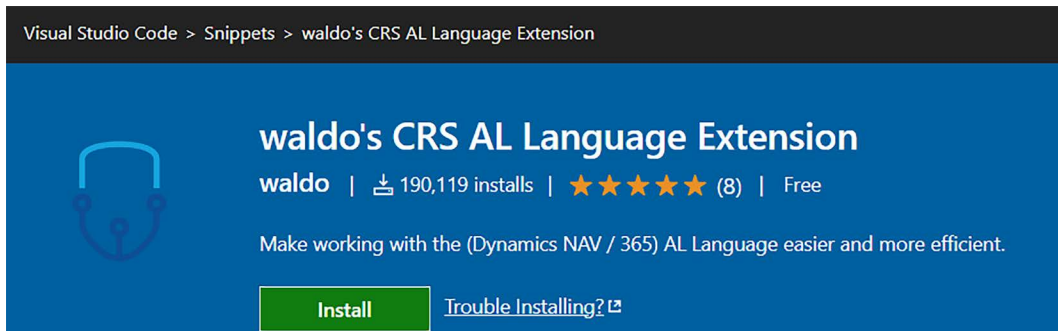


Figure 17.4: waldo's CRS AL Language Extension

The extension can be found at the following link: <https://marketplace.visualstudio.com/items?itemName=waldo.crs-al-language-extension>.

The main reason why so many people are using it is that it manages AL filename conventions: developers don't have to worry anymore about how to name their files—this extension can take care of that automatically. But it does a lot more. Let's give an overview of its functionality.

## Run objects

We all know that we can run a table or page whenever we publish an app by changing some settings in the `launch.json` file. But that's not convenient.

We need a way to be able to do the following:

- Run any object in the Windows, web, tablet, or phone client.
- Run some tools directly, such as the following:
  - Test Tool
  - Event Subscribers
  - Database Locks Page
- Run the current object that is open in the client.

The `launch.json` way of running objects isn't going to help us here.

The **waldo's CRS AL Language Extension** comes with these new commands, which can be found in the command palette (*F1*):

- CRS: Run Object (Web Client).
- CRS: Run Object (Tablet Client).
- CRS: Run Object (Phone Client).
- CRS: Run Object (Windows Client).
- CRS: Run Current Object (Web Client) (*Ctrl + Shift + R*) – this runs the object from the open file (the extension needs to be published first). You can also run this command from the status bar (**Run in Web Client**) and the context menu from the explorer.
- CRS: Run Event Subscribers Page in Web Client.
- CRS: Run Database Locks Page in Web Client.

The nice thing is it will find the settings in `launch.json` and use these to run the actual objects.

## Renaming / Reorganizing files

As mentioned earlier, this is the most widely used functionality of the tool:

1. Renaming is renaming the file.
2. Reorganizing is renaming the file AND placing it in a subfolder that matches its object type.

The essence is captured in four commands that are again available from the Visual Studio Code command palette (*F1*):

- CRS: Rename – Current File.
- CRS: Rename – All Files.

- CRS: Reorganize – Current File.
- CRS: CRS: Reorganize – All Files.

There's also a parameter, in the `settings.json` file, to rename and reorganize the file automatically when saving a `.al` file:

```
"CRS.OnSaveAlFileAction": "Rename"
```

An interesting functionality of the rename/reorganize parameter is the ability to change the patterns of filenames:

```
"CRS.FileNamePattern": "<ObjectNameShort><ObjectTypeShortPascalCase>.al",
"CRS.FileNamePatternExtensions":
"<ObjectNameShort><ObjectTypeShortPascalCase>.al",
"CRS.FileNamePatternPageCustomizations":
"<ObjectNameShort><ObjectTypeShortPascalCase>.al",
```

Here is an overview of some of the available tool settings:

- `"CRS.nstfolder"`: This is the folder of the NST.
- `"CRS.WebServerInstancePort"`: This is the port number for the web client.
- `"CRS.PublicWebBaseUrl"`: Override `Launch.json` settings with this setting if necessary to run objects from VS Code.
- `"CRS.ExtensionObjectNamePattern"`: This is the typical pattern for extension object names; if set (it's not set by default), it will perform an automatic object name for extension objects:
  - `<Prefix>`
  - `<Suffix>`
  - `<ObjectType>`
  - `<ObjectTypeShort>`: A short notation of the object type
  - `<ObjectTypeShortUpper>`: `<ObjectTypeShort>` but uppercase
  - `<ObjectId>`
  - `<BaseName>`: Weird characters are removed, and it does not include prefix or suffix
  - `<BaseNameShort>`: Does not include prefix or suffix
  - `<BaseId>`: If you want this to work, you need to put `Id` in a comment after the base name
- `"CRS.ObjectNamePrefix"`: When using the reorganize/rename commands, this setting will make sure the object name (and filename) will have a prefix:
  - Tip 1: Use as a workspace setting
  - Tip 2: Use an ending space if you want the prefix to be separated by a space
- `"CRS.ObjectNameSuffix"`: When using the reorganize/rename commands, this setting will make sure the object name (and filename) has a suffix:

- Tip 1: Use as a workspace setting
- Tip 2: Use a start space if you want the suffix to be separated by a space
- "CRS.RemovePrefixFromFilename": When using the reorganize/rename commands, this setting will remove any prefix from the filename (but keep it in the object name). Tip: Use as a workspace setting.
- "CRS.RemoveSuffixFromFilename": When using the reorganize/rename commands, this setting will remove any suffix from the filename (but keep it in the object name). Tip: Use as a workspace setting.
- "CRS.ALSubFolderName": This is the variable subfolder name. "None" means you want to disable the command to move files to a subfolder.
- "CRS.DisableDefaultALSnippets": This disables the default snippets that come with the standard AL Language Extension. When you change the setting, you need to restart Visual Studio Code twice—once to disable the snippets on activation (at that time, the snippets are still loaded), and once to not load the snippets anymore.
- "CRS.DisableCRSSnippets": This disables the CRS snippets that come with this extension. When you change the setting, you need to restart Visual Studio Code twice—once to disable the snippets on activation (at that time, the snippets are still loaded), and once to not load the snippets anymore.
- "CRS.RenameWithGit": Use `git mv` to rename a file. This keeps the history of the file but stages the rename, which you should commit separately.

## Search on Google/Microsoft Docs

A small addition, but very handy when coding, is being able to easily find documentation using two new commands in the command palette:

- CRS: Search Microsoft Docs
- CRS: Search Google

It will take the selected word, and search for that word on Google or Microsoft Docs, with Business Central as the main topic.

## Snippets

Last, but definitely not least, several snippets are included. First, there are improved versions of the Microsoft snippets:

- Removed unused triggers
- Improved tab stops
- Improved uncompileable code
- Removed default global variables

There are also new snippets that implement some default design patterns. Here are a few examples:

- `tmynotifications` (CRS): The implementation of my notifications for your own notifications

- `tassistedsetup` (CRS): The implementation of the assisted setup for your own wizards
- `tcodeunit` (CRS Method): Snippets for implementing a default encapsulated method design pattern that implements an `OnBefore` and `OnAfter` event by default

It's a good idea to explore all the snippets and familiarize yourself with them.

## Feedback to Waldo

If you have feedback, or you want to contribute to this project, then don't hesitate to fork or create issues on the repository for the CRS AL Language Extension, which you can find on GitHub at <https://github.com/waldo1001/crs-al-language-extension>.

## Waldo GitHub repo

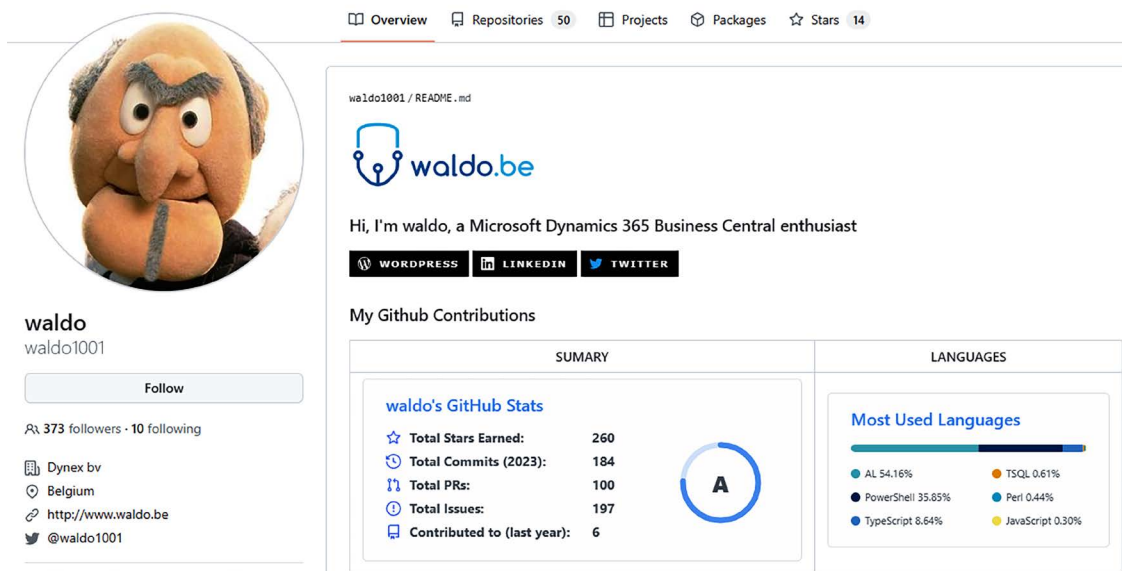


Figure 17.5: Waldo GitHub repo

As one of the most influential MVPs, Waldo is always eager to share his knowledge, insights, tips, and tricks with the Dynamics 365 Business Central community. His public GitHub repository is full of gems and goodies. Here is a list of some of them and their purposes:

- <https://github.com/waldo1001/waldo.BCPerfTool>: A useful tool that combines the power of taking AL profiler traces with different refactored samples of the same code and comparing performance results. It could also invoke an external service to gather a flame graph.

- <https://github.com/waldo1001/waldo.BCTelemetry>: A repo for AL-targeted KQL queries to analyze deadlock situations, lock timeouts, long-running queries, and AL methods. There are also queries to determine potential refactoring patterns to implement SetLoadFields (partial records) or ReadIsolation AL statements.
- <https://github.com/waldo1001/waldo.restapp>: A wrapper, a façade, for HttpClient AL statement to make AL API developer/tester life easier.

And there are way more covering different areas: Docker, PowerShell, DevOps, and so on and so forth.

## Free Visual Studio Code extensions for AL developers

When we were writing about Visual Studio Code extensions that could help developers in their daily jobs, a great blog post from Natalie Karolak appeared that we would recommend reading and providing feedback to, if you have any: <https://nataliekarolak.wordpress.com/2023/06/15/vs-code-extensions-for-al-feature-overview/>. The blog post defines the Visual Studio Code feature that is enhanced and by which extension. In this chapter, what will follow is a list of the most common ones and a brief highlight on how they are improving productivity.

### AZ AL Dev Tools/AL Code Outline (by Andrzej Zwierzchowski)

This extension has a dependency on waldo's CRS AL Language Extension (this means that if deployed, it will also install its dependency automatically). As an example of the interaction between the two extensions, this app has a wizard to create new objects, and when you finish it, it will save the file using the settings that you have for Waldo's AL Extension. It is a very useful Swiss Army knife full of tools intended to improve proficiency within AL development. It is mostly used as an alternative to standard AL Explorer in terms of browsing AL objects within extensions and symbols and it offers a list or a fabulous tree view mode to dig deep into symbols.

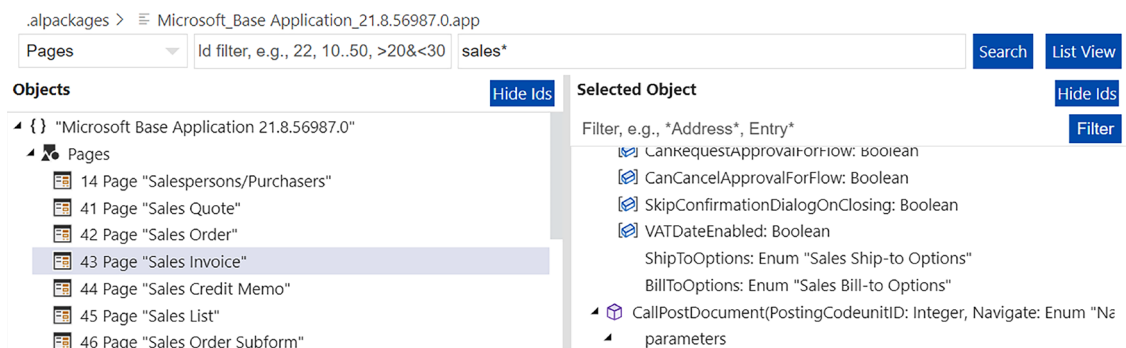


Figure 17.6: Symbol tree view

Regardless whether you are a C/AL dino-developer or not, you might find the wizard that this extension offers useful for developing new AL objects. You can easily abandon it for the more AL-ish snippets if you prefer not to use it. It offers a huge variety of code actions and commands to refactor your code and object tree. Just press **F1** to pop in the command palette and search for AZ.

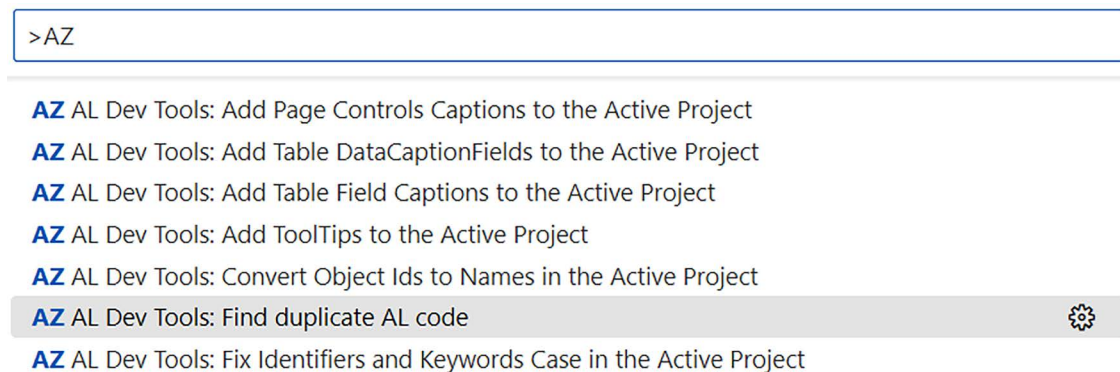


Figure 17.7: AZ actions and commands

One of my favorites is the **AZ AL Dev Tools: Show Action Images**, which creates a grid showing all the icon pictures available for actions.

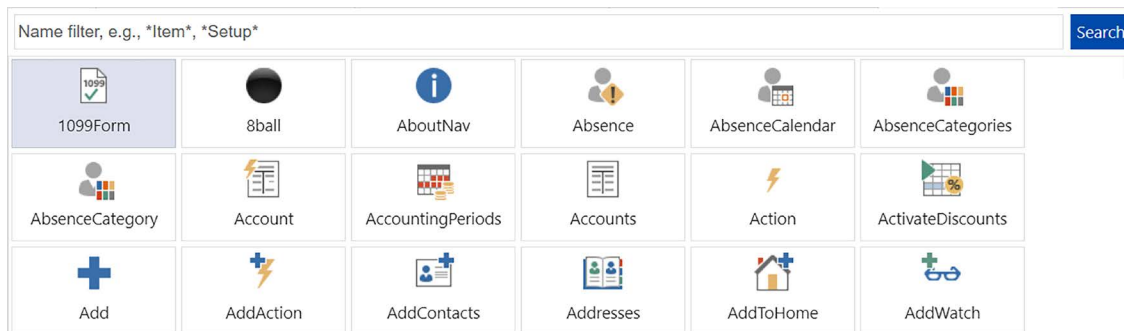


Figure 17.8: Grid of icon pictures

For his great work with this tool and his endless effort in helping the Dynamics 365 Business Central community, Andrzej Zwierzchowski – the author of this extension – has been awarded MVP for Business Applications by Microsoft.

## AL Code Actions (by David Feldhoff)

This extension has a dependency on **AZ AL Dev Tools/AL Code Outline** and boosts Visual Studio Code Actions to its max for AL developers. Code actions are marked with a light bulb when there are possible actions or corrections after an error or warning within a specific line or sentence. To learn more about it, please visit <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-code-actions>.

This extension adds very useful and fast code actions that can create procedures with parameters and method overloads at the speed of light. One thing that you would love about this app is creating new functions. Probably, you will never type a new procedure anymore, just type in a function name, with all its parameters, and the code action will create it with all the parameters for you. This is a super-boosting feature that really enhances the developer's productivity.

### **AL Object ID Ninja (by Vjeko.com)**

If you need to handle object IDs within your extension, then this extension is a must-have. We personally recommend including this extension in your code development lineup.



### **BusinessCentral.LinterCop (by Stefan Maron)**

Not only is Stefan keeping the Dynamics 365 Business Central history for Base Application updated or creating a web page where you can find useful free extensions for your ERP, but he is also famous for creating a boosted-up set of development rules (a code cop) to be integrated with the standard ones. The aim of this extension is to keep the assembly that stores the code cop always up to date. To learn more about LinterCop, please visit <https://github.com/StefanMaron/BusinessCentral.LinterCop>.



### **AL Toolbox (by Bart Permentier)**

Another toolbox full of goodies. If you are working with pragma directives and region, it is one of the *ne plus ultra*. There are a lot of interesting and powerful (code) actions that could make your development faster.



### **AL Navigator (by Waldemar Brakowski)**

Yet another super cool Visual Studio Code extension with a lot of useful code actions, mostly in the reporting area. One of the best goodies, in our opinion, is the capability of copying a report together with its layout.

## **Free Visual Studio Code extensions for AL Language side features**

There is not only AL language in the daily life of a Dynamics 365 Business Central developer. Other languages, features, technologies, or simply file handling are required to complete a solution. In such cases, there are specific extensions that might be very helpful to accomplish such daily tasks. Here is a list of some useful ones and the features that could help you to make your life easier and more productive:

- **Translations:** NAB AL Tools by NAB Solutions AB and XLIFF Sync by Rob van Bekkum.
- **GUID handling:** Create GUID by nwallace, Insert GUID by Heath Stewart, and UUID to Clipboard by Andrea Carratta.
- **TODO:** TODO Highlight by Wayou Liu and TODO Tree by Gruntfuggly.
- **Git:** GitLens by Gitkraken, Git History by Don Jayamanne, and Git Graph by mhutchie.
- **API:** Rest Client by Huachao Mao, Postmanby Postman, and Thunder Client by thunderclient.com.

- **Containers handling:** Docker by Microsoft and Docker Explorer by Jun Han.
- **Miscellaneous:** PowerShell and Live Share by Microsoft, AL Test Runner by James Pearson, Partial Diff by Ryuichi Inagaki, Bracket Select by Chunsen Wang, Code Spell Checker by Street Side Software, Bookmarks by Alessandro Fragnani, Snippet Creator by Wware Consulting, AL Structure Creator by EdySpider, Excel Viewer by Grapecity, and XML by Red Hat.

## Summary

In this chapter, we saw an interesting set of third-party tools that can help you save time when developing extensions for Dynamics 365 Business Central.

We have looked at tools developed by some of the famous names in the Microsoft Business Applications world, most notably Waldo. Some of these tools may suit your preference more than others, but the potential benefits of each one make it worthwhile giving them a go. It's also worth keeping your eyes peeled for any new tools that enter the scene.

## Leave a review!

*Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.*



*\*Limited Offer*

# 18

## Creating Generative AI Solutions for Dynamics 365 Business Central

In the previous chapter, we had an overview of a set of useful tools that can improve your development experience.

In this chapter, we want to talk about a hot topic that, over the last year, has become present in every Microsoft product: **generative AI** and **copilots**. In the Microsoft world, you have **Microsoft Copilot** (a free and open AI chatbot similar to OpenAI's ChatGPT), **Microsoft 365 Copilot** (an AI assistant integrated into M365 products), and then a set of product-specific embedded AI features, commonly known as “copilots.”

Microsoft is investing a lot in embedding these copilots in its entire suite of products and Dynamics 365 Business Central has been impacted by this change. **Copilot in Business Central** embeds different native AI features into the standard product, including marketing text generation and bank account reconciliation. You can read more about these standard features here:

- <https://learn.microsoft.com/en-us/dynamics365/business-central/ai-overview>
- <https://learn.microsoft.com/en-us/dynamics365/business-central/bank-reconciliation-with-copilot>

But that's not all. You can also create your own AI solutions by using the **Copilot developer toolkit** (a set of native objects available in the Dynamics 365 Business Central system application for interacting with Azure OpenAI). In this chapter, our focus will be on this capability – the generative AI solutions that we can create inside **Business Central (BC)**.

In this chapter, you will learn the following:

- What generative AI is and its main concepts
- Deploying and using Azure OpenAI models

- Creating an AI solution for Dynamics 365 Business Central with the Copilot developer toolkit

## Introduction to generative AI main concepts

The success of ChatGPT, developed by OpenAI, has generated a worldwide rise in interest in generative AI. As we can see from the following Google Trends data, interest in generative AI increased significantly after ChatGPT was released in November 2022:

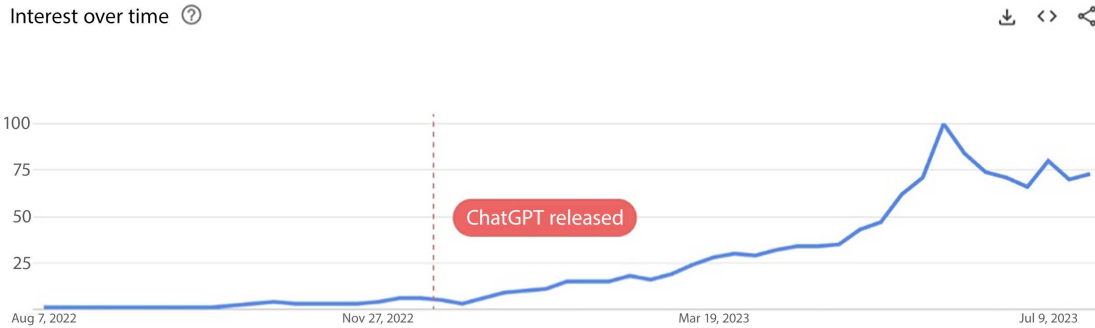


Figure 18.1: Google Trends data for ChatGPT

Let's take a glance at a high-level overview of what concepts and terminologies are important to learn in this topic. Firstly, **generative AI** is a form of artificial intelligence in which models are trained to generate new content based on natural language input. The inputs and outputs of these models can include text, images, audio, and other types of data.

Generative AI models use neural networks to identify the patterns and structures within existing data to generate new and original content.

Generative AI applications are powered by **Large Language Models (LLMs)**, like OpenAI's GPT-3.5 and GPT-4. Google's Gemini, Meta's Llama 2, and so on), which are a specialized type of machine learning model that you can use to perform **natural language processing (NLP)** tasks, like determining sentiment or otherwise classifying natural language text, summarizing text, comparing multiple text sources for semantic similarity, or generating new natural language.

LLMs can process **prompts** written in natural language. *Prompt* writing (or, to be formal, **prompt engineering**) is the process of creating input (usually text) instructing the **generative AI** model to generate the desired response. *Writing good prompts is key in generative AI*: if a prompt is not accurate, the AI response may also not be so accurate. More information about prompt engineering can be found here: <https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/prompt-engineering>.

Finally, once we start making our own copilots later in this chapter, it may be relevant to understand **tokenization**. This is the process of splitting the input and output texts (prompts) into smaller units that can be processed by the AI LLMs. A **token** is the smallest unit into which text data can be broken for an AI model to process and can be words, characters, sub-words, or symbols, depending on the type and size of the model.

The fixed context window of AI models means there's a cap on the number of tokens that can be processed in one instance. This affects the length and complexity of text the model can handle (i.e., the amount of information that you can send in a prompt to an AI model), and you should be aware that when architecting your AI solutions for Dynamics 365 Business Central, prompt size limitations are something that you will need to handle.

## Introducing Azure OpenAI Service

Azure OpenAI Service is Microsoft's cloud solution for deploying, customizing, and hosting LLMs. With Azure OpenAI Service, customers get the security capabilities of Microsoft Azure while running the same models as OpenAI. Microsoft's partnership with OpenAI enables Azure OpenAI Service users to access the latest language model innovations in a more enterprise-tailored way.

Azure OpenAI supports many models that serve different needs. These models include the following:

- **GPT-4 models** are the latest generation of **generative pretrained (GPT)** models that can generate natural language and code completions based on natural language prompts.
- **GPT-3.5 models** can generate natural language and code completions based on natural language prompts. In particular, **GPT-3.5-turbo** models are optimized for chat-based interactions and work well in most generative AI scenarios.
- **Embeddings models** convert text into numeric vectors and are useful in language analytics scenarios such as comparing text sources for similarities.
- **DALL-E models** are used to generate images based on natural language prompts. Currently, DALL-E models are in preview. DALL-E models aren't listed in the Azure OpenAI Studio interface and don't need to be explicitly deployed.

Developers can work with these models in **Azure OpenAI Studio**, a web-based environment where AI professionals can deploy, test, and manage LLMs that support generative AI app development on Azure:

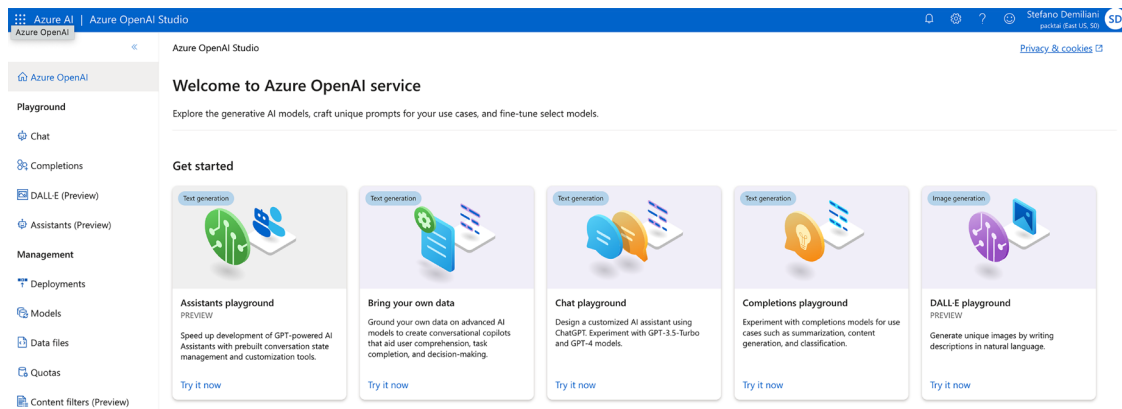


Figure 18.2: Azure OpenAI Service models

The availability of LLMs has led to the emergence of a new category of computing known as **copilots**. As we noted in the introduction, copilots are often integrated into other applications and provide a way for users to get help with common tasks from a generative AI model. Copilots are based on a common

architecture, so developers can build custom copilots for various business-specific applications and services.

Remember also that Azure OpenAI Service is not a free offering, but has a pricing policy based on models and token usage. More information can be found here: <https://azure.microsoft.com/en-us/pricing/details/cognitive-services/openai-service/>.

# Dynamics 365 Business Central and Azure OpenAI Service

Dynamics 365 Business Central has embedded generative AI features powered by Azure OpenAI Service out of the box. These AI features use a Microsoft-owned Azure OpenAI Service instance by default.

When new standard generative AI features (or so-called **copilot capabilities**) are released in Dynamics 365 Business Central during minor updates, these capabilities are optional until the next major update. To turn these features on or off (like, for example, bank reconciliation and marketing text suggestions) you need to go to the **Feature Management** page, and from there, set the **Enabled for** column to **All users** (feature active) or **None** (feature disabled):

← Feature Management

Search Analyze Edit List

Feature		Automatically enabled from	Enabled for	Get started	Current Company Status
→ Feature Update: Replace the existing Automatic Account Codes functionality with the new A...	Learn ...	Update 25.0 (Q4 2024)	None	—	Disabled
Feature Preview: Bank account reconciliation with Copilot	Learn ...	Update 24.0 (Q2 2024)	None	—	Disabled
Feature Update: Use the platform table 'Report Layout List' for adding and selecting layouts...	Learn ...	Update 24.0 (Q2 2024)	None	—	Disabled
Feature: Create AI-powered product descriptions with Copilot	Learn ...	Update 24.0 (Q2 2024)	None	—	Disabled
Feature Update: Replace the existing EU 3-Party Trade Purchase functionality with the new E...	Learn ...	Update 26.0 (Q2 2025)	None	—	Disabled
Feature Update: Enable use of new extensible exchange rate adjustment, including posting r...	Learn ...	Update 26.0 (Q2 2025)	None	—	Disabled
Feature Update: Enable use of new extensible invoice posting engine	Learn ...	Update 26.0 (Q2 2025)	None	—	Disabled
Feature: Convert user group permissions	Learn ...	Update 25.0 (Q4 2024)	None	—	Disabled

Figure 18.3: Setting the Enabled for status for copilot capabilities

When you create an AI solution for Dynamics 365 Business Central, you need to declare your copilot as a “capability” in your AL code and then register it. We’ll learn more about that later.

Using the **Copilot & AI capabilities** page, an administrator can turn individual capabilities off or on for all users as needed:

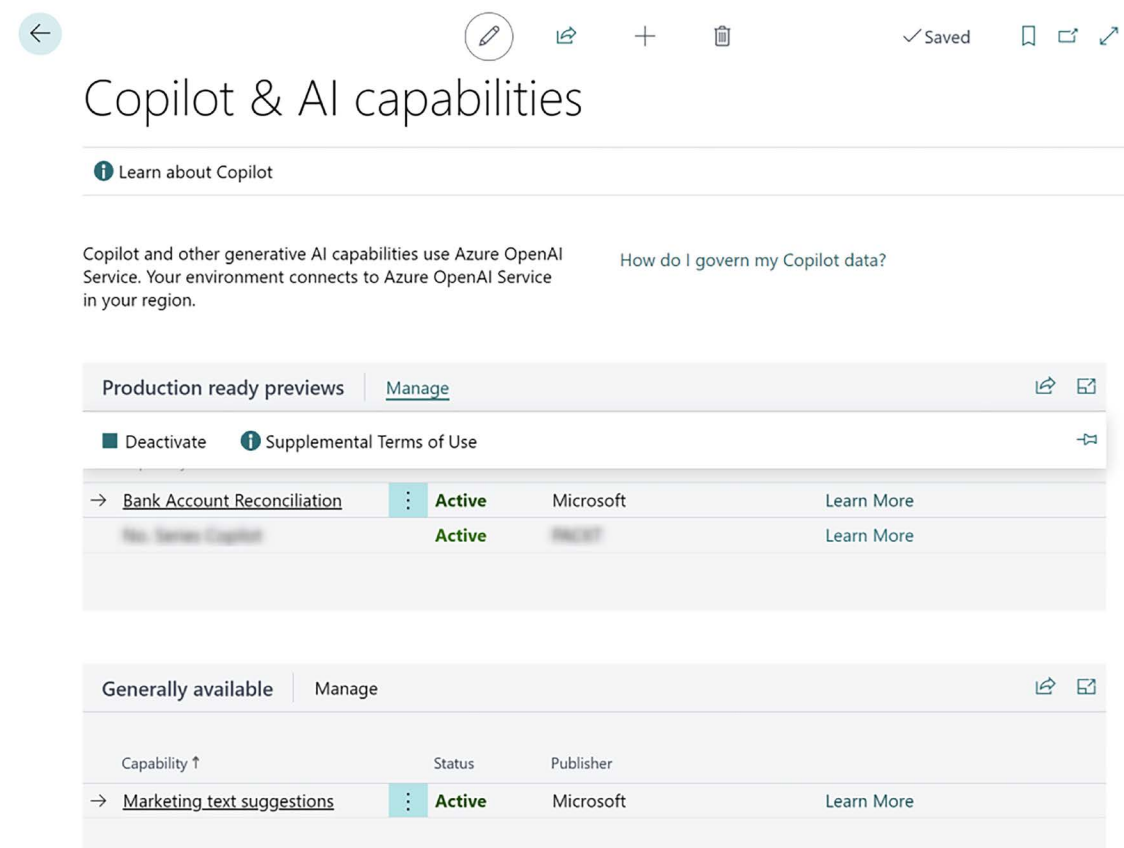


Figure 18.4: Setting the Active status for copilot capabilities

Sometimes, a Dynamics 365 Business Central environment could be in a different geography than the Azure OpenAI Service it uses (Microsoft instance). Copilot in Business Central is available in all supported Dynamics 365 Business Central regions (more details here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/compliance/apptest-countries-and-translations>) but it uses Microsoft Azure OpenAI Service, which is currently only available for Dynamics 365 Business Central in some geographical regions.

If your environment is located in one of those countries where Azure OpenAI for Business Central is not available, data from Copilot and generative AI features (prompts and data used or generated by Copilot) must be transmitted outside of your geographical region and might be processed and stored outside your compliance boundary.

In this case, an **Allow data movement** switch appears near the top of the **Copilot & AI capabilities** page, which you need to toggle on:

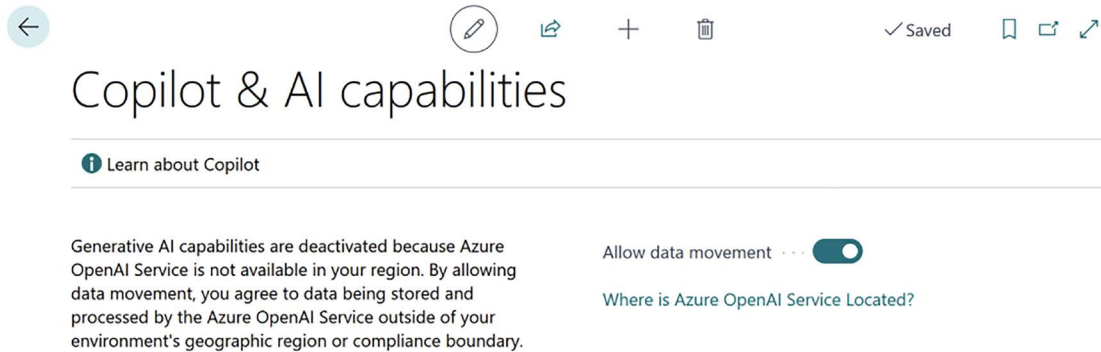


Figure 18.5: Allowing data movement

This switch will disappear when an Azure OpenAI Service instance used by Copilot is available in your environment's geographical location.

You can create custom generative AI solutions in Dynamics 365 Business Central by using a set of AL objects created for interacting with Azure OpenAI Service models and you can use your own Azure OpenAI Service models (deployed in your Azure subscription). This new set of objects permits you to have a common design pattern for AI solutions that also includes the UI and the way to interact with the AI model. This guarantees consistency of usage across applications.

## Deploy an AI model with Azure OpenAI Service

To start creating custom generative AI solutions in Dynamics 365 Business Central, you need to create your own Azure OpenAI Service instance and then deploy your own AI model.

From the Azure portal, search for **Azure AI services** and then create a new Azure OpenAI Service instance:

[Home](#) > [Azure AI services](#) | [Azure OpenAI](#) >

# Create Azure OpenAI ...

- 1 Basics
- 2 Network
- 3 Tags
- 4 Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

[Learn more](#)

## Project Details

Subscription \* ⓘ

Resource group \* ⓘ

(New) packtairg

Create new

## Instance Details

Region ⓘ

East US

Name \* ⓘ

packtai

Pricing tier \* ⓘ

Standard S0

[View full pricing details](#)

Figure 18.6: Creating a new Azure OpenAI Service instance

Verify that all networks can access your AI instance. The other available options are for situations where you want to restrict access to the Azure OpenAI Service instance only to selected virtual networks or to allow only traffic coming from private endpoints, but these are not valid options for Business Central AI solutions.

[Home](#) > [Azure AI services](#) | [Azure OpenAI](#) >

## Create Azure OpenAI ...

☒ Basics
 ☒ **2 Network**
☐ 3 Tags
 ☐ 4 Review + submit

Configure network security for your Azure AI services resource.

Type \*

- ☒ All networks, including the internet, can access this resource.
- ☐ Selected networks, configure network security for your Azure AI services resource.
- ☐ Disabled, no networks can access this resource. You could configure private endpoint connections that will be the exclusive way to access this resource.

Figure 18.7: Selecting a network type

Then click on **Create**. The provisioning of your Azure OpenAI Service instance starts.

When the deployment is completed, under **Resource Management**, select the **Model deployments** option and click on **Manage Deployments**:

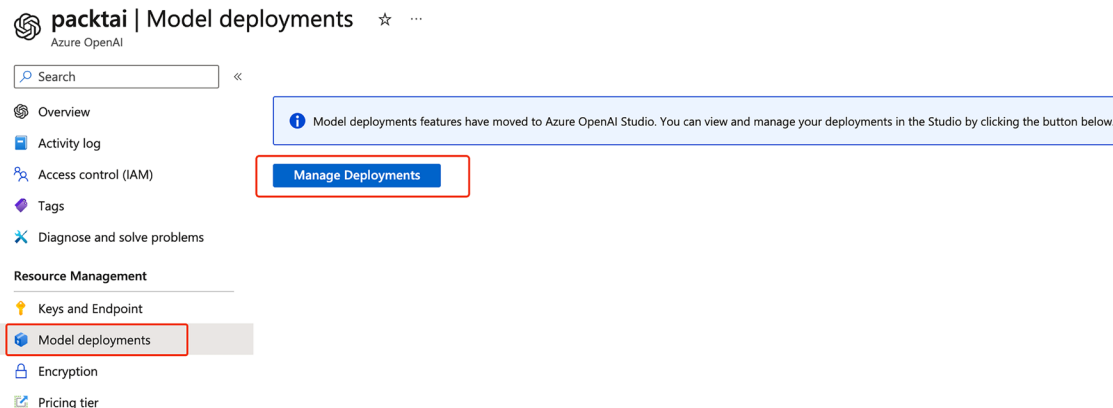


Figure 18.8: Navigating to Manage Deployments

You will be redirected to Azure OpenAI Studio, where you can create and manage your AI model deployments. Deployments provide endpoints to the Azure OpenAI Service base models, or your fine-tuned models, configured with settings to meet your needs, including the content moderation model, version handling, and deployment size.

Here, select **Deployments** and then click on **Create new deployment**:

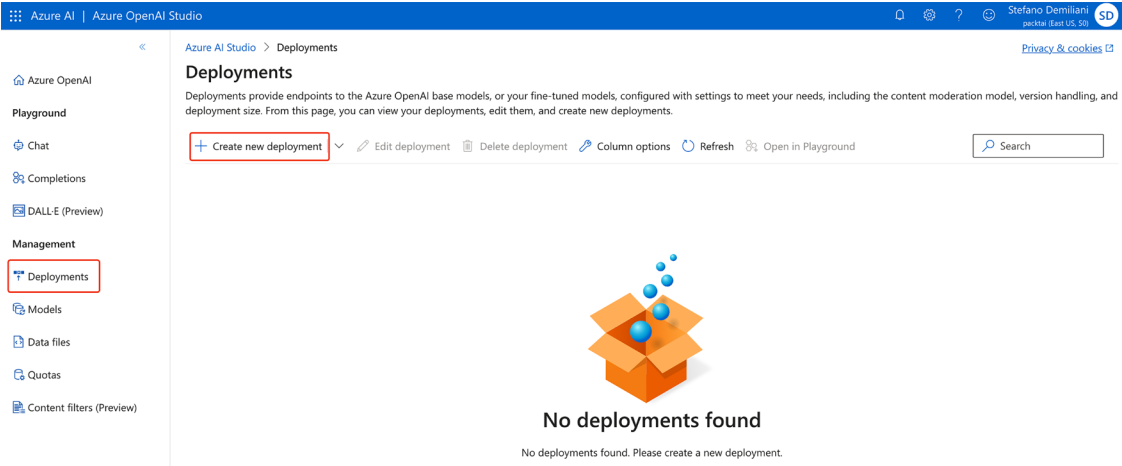


Figure 18.9: Creating a new deployment

In the **Deploy model** window, select an AI model from the list of available models and set a name for your model deployment. You can also select a particular model version or auto-update to the default version:

## Deploy model

×

Set up a deployment to make API calls against a provided base model or a custom model. Finished deployments are available for use. Your deployment status will move to succeeded when the deployment is complete and ready for use.

Select a model ⓘ

gpt-35-turbo

▼

Model version ⓘ

Auto-update to default

▼

Deployment name ⓘ

gpt-35-turbo

\*

⚙️ Advanced options >

Create

Cancel

Figure 18.10: Selecting an AI model to deploy



# Creating a generative AI solution for Dynamics 365 Business Central

One of the common tasks that a company needs to perform when starting to use Dynamics 365 Business Central is to create number series for entities (like Customer, Vendor, Item, Purchase Order, Sales Invoice, etc.). To do that, the user needs to go to the **No. Series** page and manually create a new record by setting the number series properties including defining the starting and ending numbers, specifying whether this number series will be used to assign numbers automatically, whether you can enter numbers manually instead of using this number series, whether a check that numbers are assigned chronologically must be done or not, and whether this number series allows gaps in the sequence.

Now imagine having a generative AI feature (this is our copilot) doing that for you, with something like the following button:

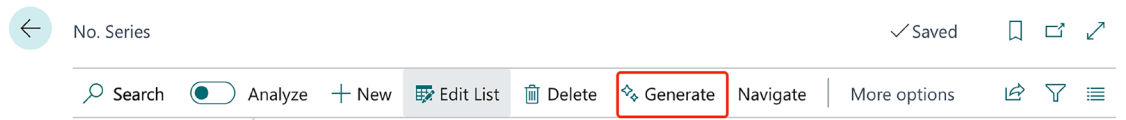


Figure 18.13: Objective of generative AI solution

By clicking the **Generate** button, our copilot's UI starts by asking you to insert the entity for which you want to create a number series:

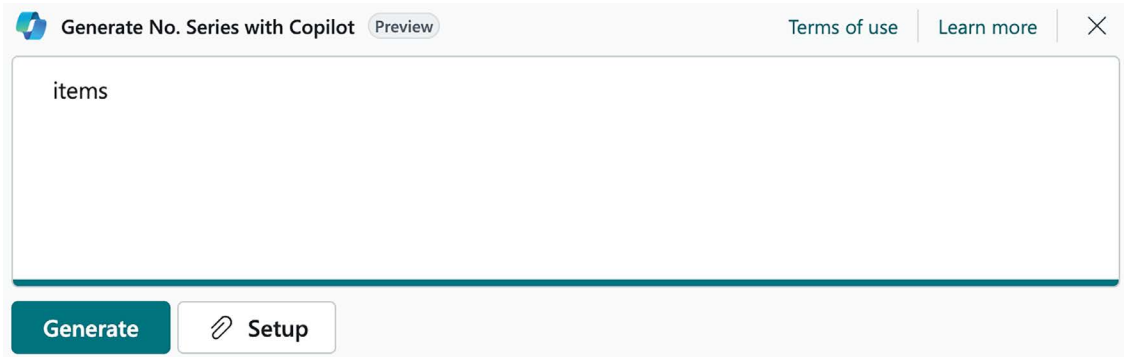


Figure 18.14: Functionality of generative AI solution

Then, by clicking on **Generate**, the **No. Series** record is automatically created for you by the AI:

items

Preview

AI-generated content may be incorrect

Terms of use

Learn more

No. Series proposals

Manage

Series Code ↑	Description	Starting No.	Increment-by No.	Ending No.	Warning No.
→ ITEMS	Item Number Series	100000	1	999999	900000

✓ Keep it

Regenerate

Cancel

Figure 18.15: Results of generative AI solution

You can also write some more details on the number series you want:

Generate No. Series with Copilot

Preview

Terms of use

Learn more

×

customer, starting from C000010 to C000999

Generate

Setup

Figure 18.16: Further functionality of generative AI solution

And our copilot will put it into action:

customer, starting from C000010 to C000999

Preview

AI-generated content may be incorrect

Terms of use

Learn more

No. Series proposals

Manage

Series Code ↑	Description	Starting No.	Increment-by No.	Ending No.	Warning No.
→ CUSTOMER		C000010	1	C000999	

✓ Keep it

Regenerate

Cancel

Figure 18.17: Further results of generative AI solution

How cool is that, having AI automation available in your Business Central solutions?

Let's see how we can create these automations (thanks to my friend Dmitry Katson (<https://katson.com/>) for the support in writing this sample).

First, we have added the copilot action to the No. Series page:

```
pageextension 61120 "PKT No. Series Ext" extends "No. Series"
{
    actions
    {
        addfirst(Promoted)
        {
            actionref("PKT Generate_Promoted"; "PKT Generate") { }
        }
        addlast("&Series")
        {
            action("PKT Generate")
            {
                Caption = 'Generate';
                ToolTip = 'Generate No. Series using Copilot';
                Image = Sparkle;
                ApplicationArea = All;
                trigger OnAction()
                var
                    NoSeriesCopilot: Page "PKT No. Series Proposal";
                begin
                    NoSeriesCopilot.LookupMode := true;
                    if NoSeriesCopilot.RunModal = Action::LookupOK then
                        CurrPage.Update();
                end;
            }
        }
    }
}
```

This action uses a new image called **Sparkle**. You have two available images for copilot actions in Dynamics 365 Business Central: **Sparkle** and **SparkleFilled**. Using these images is mandatory to respect the UI guidelines for Copilot in Business Central. Regarding the usage of these icons, Microsoft's official recommendations are the following:

- By default, use the **Sparkle** image for all copilot actions.
- If there are multiple copilot actions in an area, and one of them is considered more **important** or needs to be emphasized, then use the **SparkleFilled** icon.
- Typically, there will be zero or one **filled** action, while the rest will use the normal **Sparkle** icon.

The copilot action launches a page called PKT No. Series Proposal. This page is a new page of type PromptDialog, defined as follows:

```
page 61120 "PKT No. Series Proposal"
{
    Caption = 'Generate No. Series with Copilot';
    DataCaptionExpression = GenerationIdInputText;
    PageType = PromptDialog;
    IsPreview = true;
    Extensible = false;
    ApplicationArea = All;
    Editable = true;
    SourceTable = "Name/Value Buffer";
    SourceTableTemporary = true;
    InherentPermissions = X;
    InherentEntitlements = X;

    layout
    {
        area(PromptOptions)
        {
        }

        area(Prompt)
        {
            field(InputText; InputText)
            {
                ShowCaption = false;
                Multiline = true;
                ApplicationArea = All;
                trigger OnValidate()
                begin
                    CurrPage.Update();
                end;
            }
        }

        area(Content)
        {
            part(ProposalDetails; "PKT No. Series Proposal Sub")
            {
                Caption = 'No. Series proposals';
            }
        }
    }
}
```

```
        ShowFilter = false;
        ApplicationArea = All;
        Editable = true;
        Enabled = true;
        SubPageLink = "Generation Id" = field(ID);
    }
}
}
actions
{
    area(SystemActions)
    {
        systemaction(Generate)
        {
            Tooltip = 'Generate no. series';
            trigger OnAction()
            begin
                GenerateNoSeries();
            end;
        }
        systemaction(Regenerate)
        {
            Caption = 'Regenerate';
            Tooltip = 'Regenerate no. series';
            trigger OnAction()
            begin
                GenerateNoSeries();
            end;
        }
        systemaction(Attach)
        {
            Caption = 'Setup';
            trigger OnAction()
            begin
                Page.Run(Page::"PKT No. Series Copilot Setup");
            end;
        }
        systemaction(Cancel)
        {
            ToolTip = 'Discards all suggestions and dismisses the dialog';
        }
    }
}
```

```

        systemaction(Ok)
        {
            Caption = 'Keep it';
            ToolTip = 'Accepts the current suggestion and dismisses the
dialog';
        }
    }
}

```

The PromptDialog page type allows you to integrate copilot capabilities into your custom scenarios. You can use the PromptDialog page type to create generative AI experiences with the Copilot in Business Central look and feel, which includes signature visuals and built-in safety controls for customers.

The PromptDialog page has a specific page syntax, which includes new area and action controls. When defining a PromptDialog page, you need to specify a set of properties. The most important is the PromptMode property (by default set to Prompt), which is the starting prompt mode for the Copilot UI.

The PromptMode property can then be changed at runtime via code by using the CurrPage.PromptMode property before the page is opened. The other options for PromptMode are as follows:

- **Generate:** Triggers generating the output of the Copilot interaction
- **Content:** Shows the output of the copilot interaction

The PromptDialog page type has three main areas:

- **Prompt area:** This is the input area for your copilot and accepts any control, except repeater controls.
- **Content area:** This is the output of your copilot, and accepts any control, except repeater controls.
- **PromptOptions area:** This is the input options, and only accepts option fields.

The Generate action calls a GenerateNoSeries procedure that:

- Creates a standard prompt in code. The prompt is complex and provides instructions to the AI model on how to receive the question and how we want the response to be formatted (we want JSON here).
- Calls the Azure OpenAI Service model (completion API), passing the prompt.
- Reads the model response by parsing the JSON.
- Prints the response to the user.

Here is how the prompt for the AI model is defined:

```

local procedure GetSystemPrompt(): Text
var
    SystemPrompt: TextBuilder;
begin
    SystemPrompt.AppendLine('You are `generateNumberSeries` API');

```

```

        SystemPrompt.AppendLine();
        SystemPrompt.AppendLine('Your task: Generate No. Series for the next
entities:');
        SystemPrompt.AppendLine('');
        ListAllTablesWithNoSeries(SystemPrompt);
        SystemPrompt.AppendLine('');
        SystemPrompt.AppendLine();
        SystemPrompt.AppendLine('User might add additional instructions on how
to name series.');
```

```

        SystemPrompt.AppendLine('Try to fulfill them.');
```

```

        SystemPrompt.AppendLine();
        SystemPrompt.AppendLine('IMPORTANT!');
```

```

        SystemPrompt.AppendLine('Don't add comments.');
```

```

        SystemPrompt.AppendLine('Fill all fields.');
```

```

        SystemPrompt.AppendLine('Always respond in the next JSON format:');
```

```

        SystemPrompt.AppendLine('');
```

```

        SystemPrompt.AppendLine('[');
        SystemPrompt.AppendLine('    {');
        SystemPrompt.AppendLine('        "seriesCode": "string (len 20)",');
        SystemPrompt.AppendLine('        "lineNo": "integer",');
        SystemPrompt.AppendLine('        "description": "string (len 100)",');
        SystemPrompt.AppendLine('        "startingNo": "string (len 20)",');
        SystemPrompt.AppendLine('        "endingNo": "string (len 20)",');
        SystemPrompt.AppendLine('        "warningNo": "string (len 20)",');
        SystemPrompt.AppendLine('        "incrementByNo": "integer"');
        SystemPrompt.AppendLine('    }');
        SystemPrompt.AppendLine(']');
```

```

        SystemPrompt.AppendLine('');
        SystemPrompt.AppendLine('If you can't answer or don't know the
answer, respond with: []');
```

```

        SystemPrompt.AppendLine('Your answer in a JSON format: []');
```

```

        exit(SystemPrompt.ToText());
    end;
```

The AI model is called by using the following AL code:

```

[NonDebuggable]
    internal procedure GenerateNoSeries(var SystemPromptTxt: Text; InputText:
Text): Text
    var
        AzureOpenAI: Codeunit "Azure OpenAi";
        AOAIDeployments: Codeunit "AOAI Deployments";
```

```

    AOAIOperationResponse: Codeunit "AOAI Operation Response";
    AOAIChatCompletionParams: Codeunit "AOAI Chat Completion Params";
    AOAIChatMessages: Codeunit "AOAI Chat Messages";
    CompletionAnswerTxt: Text;
begin
    if not AzureOpenAI.IsEnabled(Enum::"Copilot Capability"::"PKT No.
Series Copilot") then
        exit;

    AzureOpenAI.SetAuthorization(Enum::"AOAI Model Type"::"Chat
Completions", GetEndpoint(), GetDeployment(), GetSecret());
    AzureOpenAI.SetCopilotCapability(Enum::"Copilot Capability"::"PKT No.
Series Copilot");
    AOAIChatCompletionParams.SetMaxTokens(MaxOutputTokens());
    AOAIChatCompletionParams.SetTemperature(0);
    AOAIChatMessages.AddSystemMessage(SystemPromptTxt);
    AOAIChatMessages.AddUserMessage(InputText);
    AzureOpenAI.GenerateChatCompletion(AOAIChatMessages,
AOAIChatCompletionParams, AOAIOperationResponse);
    if AOAIOperationResponse.IsSuccess() then
        CompletionAnswerTxt := AOAIChatMessages.GetLastMessage()
    else
        Error(AOAIOperationResponse.GetError());

    exit(CompletionAnswerTxt);
end;

```

The code uses the Azure OpenAI Service AL objects exposed by the Copilot developer toolkit for authenticating to Azure OpenAI Service, setting the model temperature parameter (used in AI to control the randomness of the output of a model, where a lower temperature results in more predictable output, while a higher temperature results in more random output) and other request parameters and then calling the Completion API.

With the **Completions** operation, the model will generate one or more predicted **completions** based on a provided prompt.

Before calling the Azure OpenAI Service model from AL code, you need to register your copilot capabilities in Business Central:

```

AzureOpenAI.SetCopilotCapability(Enum::"Copilot Capability"::"PKT No. Series
Copilot");

```

Copilot custom capabilities can be added by extending the Copilot Capability enum:

```

enumextension 61120 "PKT No. Series Copilot Cap." extends "Copilot Capability"

```

```

{
    value(50100; "PKT No. Series Copilot")
    {
        Caption = 'No. Series Copilot';
    }
}

```

Registration of your custom copilot capabilities should be done in an Install codeunit like the following:

```

codeunit 61120 "PKT No. Series Copilot Install"
{
    Subtype = Install;

    trigger OnInstallAppPerDatabase()
    begin
        RegisterCapability();
    end;

    local procedure RegisterCapability()
    var
        CopilotCapability: Codeunit "Copilot Capability";
        EnvironmentInformation: Codeunit "Environment Information";
        LearnMoreUrlTxt: Label 'https://www.demiliani.com', Locked = true;
    begin
        if EnvironmentInformation.IsSaaS() then
            if not CopilotCapability.IsCapabilityRegistered(Enum::"Copilot Capability"::"PKT No. Series Copilot") then
                CopilotCapability.RegisterCapability(Enum::"Copilot Capability"::"PKT No. Series Copilot", LearnMoreUrlTxt);
    end;
}

```

When creating copilots for your apps, you should always respect the standard Copilot patterns:

- Never automatically save proposed data to the database but instead show it in a temporary table.
- Give the user the possibility to accept or discard the response.
- If the AI-generated proposal is accepted by the user, save it in the database.

This is exactly what we do in this application. The proposal is stored in a temporary table and displayed in a `ListPart` object and the user has the options of keeping the proposal, regenerating a new proposal, or discarding it:

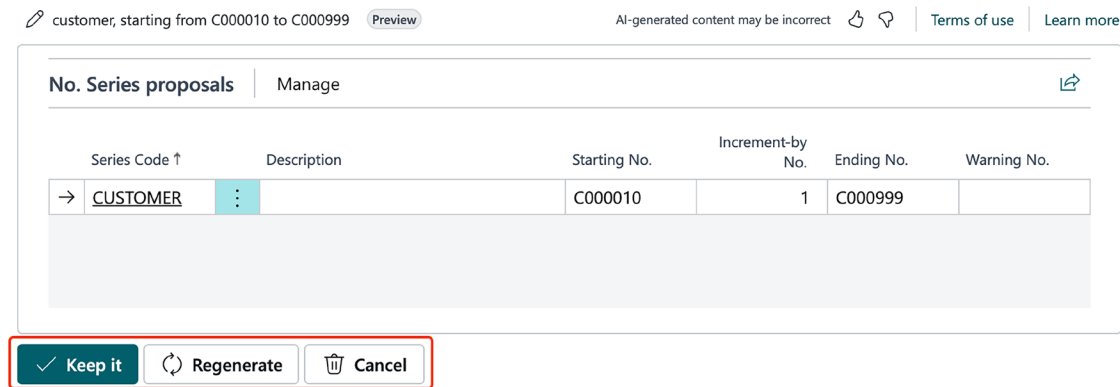


Figure 18.18: Testing the generative AI solution

If the user keeps the proposal, the data is saved into Dynamics 365 Business Central as a new number series, otherwise not.

The complete source code of this AI application is available in the book's GitHub repo: <https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central-Second-Edition>.

## Summary

In this chapter, we talked about generative AI solutions in Dynamics 365 Business Central. After explaining the main concepts related to generative AI and Azure OpenAI, we saw how to create a new Azure OpenAI instance and a new AI deployment model. Then we explained how to create a custom AI solution for Dynamics 365 Business Central by using the Copilot developer toolkit objects in AL and the toolkit's rules.

Now that you have reached the end of this chapter and explored the source code, you will be able to start adding AI capabilities to your ERP solutions.

This chapter marks the end of this edition of *Mastering Dynamics 365 Business Central*. By following this book, you've embraced all that you need to know in order to start creating advanced solutions for Dynamics 365 Business Central. You now have all the knowledge needed to become a master. Good luck!

## Leave a review!

*Enjoyed this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.*



*\*Limited Offer*





packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

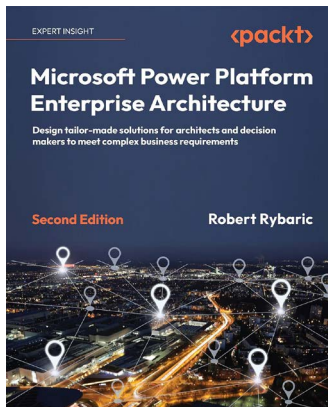
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.



# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

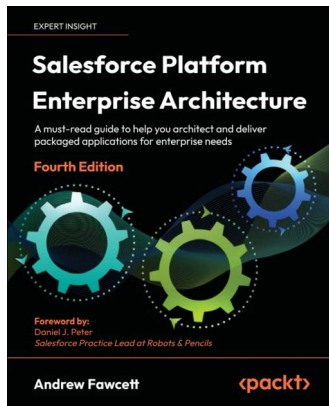


## Microsoft Power Platform Enterprise Architecture - Second Edition

Robert Rybaric

ISBN: 9781804612637

- Understand various Microsoft Dynamics 365 CRM, ERP, and AI modules for creating Power Platform solutions
- Combine Power Platform capabilities with Microsoft 365 and Azure
- Find out which regions, staging environments, and user licensing groups need to be employed when creating enterprise solutions
- Implement sophisticated security by using various authentication and authorization techniques
- Extend Microsoft Power BI, Power Apps, and Power Automate to create custom applications
- Integrate your solution with various in-house Microsoft components or third-party systems using integration patterns
- Migrate data using a variety of approaches and best practices

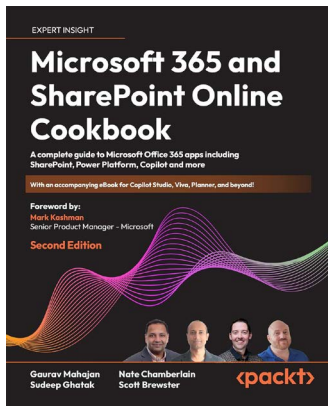


## Salesforce Platform Enterprise Architecture - Fourth Edition

Andrew Fawcett

ISBN: 9781804619773

- Create and deploy packaged apps for your own business or for AppExchange
- Understand Enterprise Application Architecture patterns
- Customize the mobile and desktop user experience with Lightning Web Components
- Manage large data volumes with asynchronous processing and big data strategies
- Learn how to go beyond the Apex language, and utilize Java and Node.js to scale your skills and code with Heroku and Salesforce Functions
- Test and optimize Salesforce Lightning UIs
- Use Connected Apps, External Services, and Objects along with AWS integration tools to access off platform code and data with your application



## Microsoft 365 and SharePoint Online Cookbook - Second Edition

Gaurav Mahajan , Sudeep Ghatak , Nate Chamberlain, Scott Brewster

ISBN: 9781803243177

- Collaborate effectively with SharePoint, Teams, OneDrive, Delve, Search, and Viva
- Boost creativity and productivity with Microsoft Copilot
- Develop and deploy custom applications using Power Apps
- Create custom bots using Power Virtual Agents (Copilot Studio)
- Integrate with other apps, automate workflows and repetitive processes with Power Automate/Desktop (RPA)
- Design reports and engaging dashboards with Power BI
- Utilize Planner, To Do, and gather feedback with polls and surveys in Microsoft Forms
- Experience seamless integration in the mobile platform

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share your thoughts

Now you've finished *Mastering Microsoft Dynamics 365 Business Central, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Index

## A

### **access modifiers, in AL**

- using 198-201

### **access token 448**

### **actions**

- handling, on errors 206, 207
- promoting 158, 159

### **Activity bar, Visual Studio Code 25**

- Command Palette 25, 26
- Manage button 25

### **aggregated signals 379, 380**

### **AI model**

- deploying, with Azure OpenAI 626-630

### **aka.ms links related to Business Central**

- reference link 13

### **AL Code Actions 618, 619**

### **AL coding guidelines**

- working on 103-106

### **alerts 394-397**

### **AL Explorer 54, 55**

### **AL Extension Pack 611, 612**

### **AL-Go for GitHub 512**

- branching strategies 517
- per-tenant extension, creating with 514-517
- templates 513
- URL 511
- using, for AppSource development 547-550

### **AL Language 65**

- extension, running in debug mode 347-350
- namespaces 239-242
- XML and JSON files, handling 214-218

### **AL Language extension 21, 39-42**

- app.json 45
- basics 69, 70
- codeunit object definition 84, 85
- download link 39
- enum object definition 94-97
- event object definition 85-89
- launch.json file 42-44
- objects, defining with snippets 70-73
- page extension object definition 82-84
- page object definition 77-80
- profile object definition 97
- query object definition 91-94
- settings 56, 57
- table extension object definition 80-82
- table object definition 74-77
- workspace 70
- XMLport object definition 89-91

### **AL Language side features**

- free Visual Studio Code extensions 619

### **AL Navigator 619**

### **AL Object ID Ninja 619**

### **AL profiler 364, 365**

### **AL project structure**

- best practices 98-100

### **AL Toolbox 619**

### **APIs**

- implementing, for custom entity 460-466
- implementing, for existing entity 467-471
- namespaces 439
- OData protocol, using for 438, 439
- versioning 439
- webhook supported 439

**app.json file 45**

- application parameter 46
- attributes 46, 47
- base/system application dependencies 45
- condensed app.json file, example 48
- features parameter 49
- parameters 50
- platform property 46
- resourceExposurePolicy parameter 49
- runtime 46

**Application Insights 377, 394****application settings 488****AppSource 3, 101****AppSource development**

- AL-Go for GitHub, using for 547-550

**Artificial Intelligence (AI) 14****asynchronous patterns**

- performances, testing 433-436
- performances, validating 433-436
- running 430
- StartSession, using in AL code 430-432
- Task Scheduler, using in AL code 432

**attachments 249**

- handling 250, 251

**authentication token**

- acquiring, from Microsoft Entra ID 448-450

**AZ AL Dev Tools/AL Code Outline 617, 618****Azure**

- storage account, creating 267, 268

**Azure Blob Storage 266, 275**

- blob 266
- container 266
- Dynamics 365 Business Central attachments, handling 270-274
- storage account 266
- using, from AL code 268-270

**Azure Files 275****Azure file shares**

- using, from AL 275-278

**Azure Functions 487, 502**

- benefits 488
- components 488
- execution modes 489
- functions, creating 489-500
- pricing plans 488
- use cases 487
- using 501-504

**Azure Functions app 488****Azure Functions Core tools 490****Azure Functions runtime 488****Azure Key Vault 236**

- references 50, 237
- using, in AL extensions 236-238

**Azure Logic Apps 504-506**

- consumption tier 504
- reference link 504
- standard tier 505
- workflows, creating 506-509

**Azure OpenAI 623, 624**

- AI model, deploying with 626-630
- and Dynamics 365 Business Central 624-626
- DALL-E models 623
- embeddings models 623
- GPT-3.5 models 623
- GPT-4 models 623

**B****BCAgent 346****bindings 489****blob fields**

- text data, reading and writing from 251

**branching strategies 517**

- feature/developer branches 518
- master-only branch 518
- other strategies 519
- release branching 519

**breadcrumbs 35**

**breakpoint** 347

**Business Central connector  
for Power Platform** 554

**BusinessCentral.LinterCop** 619

**business events** 66, 85

## C

**carriage return (CR)** 24

**CentralQ**

URL 345

**Certified for Microsoft Dynamics Program  
(CfMD)** 11

**Chrome DevTools Protocol** 368-370

**CI/CD pipeline**

handling 526-528

**cloud-ready printing** 328, 329

**Cloud Solution Provider (CSP)** 1

**code analyzers** 58-60

**code block folding** 34

**code editor, Visual Studio Code** 23, 24

**code extensions** 66

**codeunit** 84, 85

access level 200

installing 162

upgrading 162-166

versus processing-only report 282

**cold start** 488

**collectible errors**

handling 206

using 203, 204

**columnstore indexes, on SQL**

reference link 420

**comment lines** 33

**CommitBehavior** attribute

using 427

**Completions operation** 638

**components, Azure Functions**

application settings 488

Azure Functions app 488

Azure Functions runtime 488

bindings 489

scale controller 488

triggers 489

**control add-in** 222

definition 222

PDF-Viewer control add-in, creating 223-227

working with 223

**copilot capabilities** 624

**Copilot developer toolkit** 621

**Copilot in Business Central** 621

**copilots** 623

**Create, Read, Update, and Delete (CRUD)** 438

**custom API**

creating, in Dynamics 365 Business Central 459

**Customer Category implementations** 114

codeunit definition 129-132

event subscribers 132, 133

pageextension definition 126-129

pages definition 118-124

tableextension definition 124-126

tables definition 114-118

**customer Production environment**

registering, for manual deployment 544-546

**customer sandbox environment**

registering, for continuous

deployment 539-542

**custom signals** 414-417

## D

**dashboards** 397-402

**data access layer**

defining 419-423

efficient pages and reports, writing 427, 428

- performant installation and upgrade AL codeunits, writing 428, 429
- table extension changes, from Dynamics 365 Business Central version 23 423
- transaction isolation level, setting in AL code 424-426
- database events** 66
- dataset** 285
- DataTransfer data type** 428
- DataTransfer object**
  - limitations 167
- Dataverse** 595
- debugger sidebar** 352
  - breakpoints 354
  - callstack 354
  - Database Statistics tab 352, 353
  - VARIABLES section 352
  - Watch section 354
- dedicated API pages** 438
- DeleteAll** 421
- delimiter matching** 33
- Denial-of-Service (DoS) attacks** 472
- dependencies**
  - handling, between applications 533-538
- dependent extension**
  - writing 183-188
- development of localization solution**
  - reference link 3
- dimensions** 378
  - custom 378
  - general 378
- Directions EMEA** 16
- DownloadFromStream method** 247, 248
  - reference link 247
- Dynamics 365 APIs**
  - operational limits 471

- Dynamics 365 Business Central APIs**
  - OAuth authentication, configuring for 439
  - OData batch calls, using with 476-482
- Dynamics 365 Business Central App Usage Analytics** 410
- Dynamics 365 Business Central customization**
  - developing 111-114
- Dynamics 365 Business Central developers**
  - skills 17
- Dynamics 365 Business Central events**
  - exposing, to Dataverse 603-607
- Dynamics 365 Business Central Functional Consultant Associate**
  - reference link 18
- Dynamics 365 Business Central integration**
  - Power Apps app, creating with 562
- Dynamics 365 Business Central standard APIs**
  - using 450-458
- Dynamics 365 Business Central Usage Analytics** 409

## E

- email printers**
  - enabling 329, 330
- entity data model (EDM)** 451
- enums** 94-97
- error handling**
  - with, TryFunctions 202, 203
- errors**
  - actions, handling 206, 207
- event object** 85-89
- event publisher** 85
- event publisher function** 66
- events** 66, 175-182, 430
  - business events 66
  - database events 66
  - global events 67

integration events 67

page events 66

performance 430

**event subscriber 87**

**execution modes, Azure Functions**

in-process 489

isolated 489

**Expense app, with offline capability 563-565**

canvas app project, creating 565, 566

controls and business logic, adding to canvas app 569-571

data connection, establishing to Dynamics 365 Business Central 566, 567

offline capabilities, adding 582-591

Power Automate flow, calling to execute actions in app 593-595

working with Dynamics 365 Business Central data, from canvas app 568

**extensibility 173**

**extensible code**

need for 173-175

protection 193

**extensions 66**

**extension symbols 51**

## F

**fast-forward (ff) merge 521**

**field**

access level 199

**files**

handling, with AL 243-249

**Find All References feature 37**

**FindOrphans method 253**

**first-party extensions 15**

**free Visual Studio Code extensions**

for AL Language side features 619

## G

**generative AI 622**

**generative AI solution**

creating, for Dynamics 365 Business Central 631-640

**gift campaign implementations 133**

codeunit definition 139-144

page definition 135-139

table definition 133-135

**Git flow 519, 520**

**GitHub**

enterprise accounts 512

organization accounts 512, 513

personal accounts 512

**GitHub Copilot**

for AL developers 61-63

reference link 63

**GitHub flow 520**

**Git in Visual Studio Code 523**

GUI 524, 525

workflow 525

**GitLens extension 523**

**Git merge strategies 521**

fast-forward merge 521

rebase command 522, 523

squash commit, using 521

**global events 67**

**Globally Unique Identifier (GUID) 380**

**Go to definition feature 36**

**grouping 296**

## H

**Handled pattern 176-183**

**Headline, role centers**

customizing 211-214

**I**

**immutable keys** 197, 198

**InputStream** method 253

**in-client performance profiler**

designing 371

using 371-375

**Independent Software Vendor (ISV)** 14

**inside symbols** 52-54

**InStream** data type

reference link 245

**Integrated Development Environments (IDEs)** 22

**integration events** 67, 86

**Intellectual Property (IP)** 349

**IntelliSense** 36

**interfaces** 188

in AL 188-193

**Isolated Storage** 218, 262-265

data, managing 219

DataScope option 218

example 219, 220

secret management 221

**IsolationLevel** 424

**Item Ledger Entry (ILE) table** 421

**J**

**JSONArray**

reference link 258

**JSON document**

handling, with AL 214-218, 258-260

**JsonObject**

reference link 258

**K**

**Kusto Query Language (KQL)** 384

functions 387-393

log analysis 384, 385

operators 387

statements 386

**L**

**Large Language Models (LLMs)** 622

**launch.json** file 42-44, 363

attributes 42-44

**line feed (LF)** 24

**logs**

debugging, in verbose mode 350, 351

**M**

**Master** 14

**MB-820** 19

**Media** data type 252

using, in AL code 252, 253

**MediaSet** data type 252, 253

**MiBuSo** 610

**Microsoft 365 Copilot** 621

**Microsoft Collaborate documentation**

reference link 14

**Microsoft Copilot** 621

**Microsoft Dynamics 365 Business Central** 1, 2

considerations 8-10

evolution 3-8

future perspective 16-19

**Microsoft Entra ID**

application permissions, setting 442-444

application, registering 440-442

application registration, in Dynamics 365  
Business Central 445-448

authentication token, acquiring from 448-450

client secret, creating for registered

application 444, 445

**Microsoft GitHub AIAppExtensions** labels 68

**Microsoft Universal Print**

- alternatives 345, 346
- connectors list 332
- document conversion 334
- installation wizard 336
- Integration extension 335
- printer content type, setting 333
- printer pop-up, sharing 334
- printer properties 337
- printers 330
- printer share name, setting 334
- printers list 332
- report printing 338-345
- shared printer access, setting 334

**mini-map 35****ModifyAll 421****module, changing in System Application**

- reference link 15

**Most Valuable Professionals (MVPs) 14****multiple active result sets (MARS) 420****multiple cursors 34****N****namespaces, AL language 239-242****naming guidelines**

- in AL 101-103

**natural language processing (NLP) 622****Navision 3****notifications, Dynamics 365 Business Central 227-230****O****OAuth authentication**

- configuring, for Dynamics 365 Business Central APIs 439

**object ranges**

- in AL 101-103

**objects**

- defining, with snippets 70-73

**obsoleting code**

- reference link 50

**OData batch calls**

- using, with Dynamics 365 Business Central APIs 476-482

**OData bound actions**

- using 472-474

**OData protocol**

- using, for APIs 438, 439

**OData unbound actions**

- using 474-476

**open source and social networks**

- role 12-16

**operational challenges with many companies per environment**

- reference link 9

**Operational limits for Business Central online**

- reference link 8

**OutStream data type**

- reference link 245

**P****page background tasks 230**

- creating 232
- example 232-236
- properties 232

**page events 66****page extension object 82-84****page object 77-80**

- properties 78

**page views**

- creating 160-162

**pagextension object 202****panels area, Visual Studio Code 30, 31**

- DEBUG CONSOLE 32

OUTPUT 31  
 PROBLEMS 31  
 TERMINAL 32  
**partner telemetry**  
   enabling, in Dynamics 365 Business Central  
     online 381-384  
**PDF-Viewer control add-in**  
   creating 223-227  
**peek definition** 37, 38  
**performance profiling** 363-368  
   Chrome DevTools Protocol 368-370  
   Sampling Interval, setting for in-client  
     performance profiling 371-375  
**performance test app**  
   adding, to repository 546, 547  
**Performance Toolkit extension** 433, 434  
   reference link 8  
**permission sets**  
   defining, in AL 169-171  
**Persistent Blobs**  
   using 260, 261  
**Personally Identifiable Information (PII)** 380  
**Power Apps app**  
   creating, with Dynamics 365 Business Central  
     integration 562  
**Power Automate** 505, 553  
**Power Automate, with Dynamics 365 Business Central**  
   default image, adding to Customer  
     record 554-559  
   selected invoice, exporting as PDF  
     to OneDrive 559-562  
**Power BI telemetry apps** 409-413  
**Power Platform** 553  
**pricing plans, Azure Functions**  
   App Service plan 488  
   consumption plan 488  
   premium plan 488

**Printer Management page** 328

**Printer Selections page** 328

**Printix**

  URL 345

**PrintNode**

  URL 345

**procedures**

  access level 200

**processing-only report**

  versus codeunit 282

**profile object** 97

**prompt engineering** 622

**prompts** 622

**protected variables** 201

**publisher** 66

**Pull Request (PR)** 15

## Q

**QRCode** 489

**query objects** 91-94, 428

## R

**ReadIsolation property** 424

**read-only database replica**

  using 471

**rebase command** 522, 523

**Registered Solution Program (RSP)** 11

**regular expressions (regexes)** 28

**release** 543

  creating, for application 543, 544

**report, creating**

  database images, adding 300-302

  dataset, designing 285-288

  Excel layout, adding 308-312

  group table, creating for table control 296-299

  RDL report header, creating 290-292

- simple RDL layout, creating 290
- simple request page, building 299
- table control, adding to RDL
  - report body 293-295
- Word layout, adding 303-307
- Report Definition Language (RDL)**
  - layouts 35, 281
  - developing, options 284
- report extension object** 281, 284
  - advantage 284
  - basic example 313-317
  - drawback 284
- report object** 281
  - anatomy 281-283
- report printing** 338-345
  - flow chart 338
- reports** 282
  - cloning 317-322
  - feature limitations, on developing RDL or Word
    - layout document reports 322, 323
  - performance considerations 323, 324
- repository**
  - performance test app, adding 546, 547
- Resources for Partners**
  - reference link 18
- RESTful APIs**
  - reference link 439
- retry policy** 472
- Returns On Investment (ROIs)** 16
- role centers** 207
  - creating 208, 209
  - extending 208-210
  - Headline, customizing 211-214
  - page structure 208
- S**
- scale controller** 488
- self-hosted GitHub runner**
  - setting up 529-533
- service operations**
  - reference link 9
- SetLoadFields** 422
- sidebar, Visual Studio Code** 26
  - DEBUG 29
  - EXPLORER 26, 27
  - EXTENSIONS 29, 30
  - SEARCH 28
  - SOURCE CONTROL 28
- signals** 377
  - aggregated signals 379, 380
  - dimension 378, 379
  - example 381
  - Microsoft's recommendations 380, 381
- snapshot debugging** 358-362
- snippets** 70
  - used, for defining objects 70-72
- Software-as-a-Service (SaaS)** 1
- Software Lifecycle Policy and Dynamics** 365
  - Business Central On-Premises Updates
    - reference link 5
- source code management (SCM) tool** 523
- Sparkle** 633
- SQL data layer** 420
- squash commit** 521
- status bar, Visual Studio Code** 24
- storage account**
  - creating, in Azure 267, 268
- storage capacity** 265
- Storage Service Authorization interface** 275
- streams** 245
- subscriber** 67
- SumIndexField Technology (SIFT)** 420
- symbols** 38, 51, 52
  - extension symbols 51

inside symbols 52-54

system symbols 51

**system application 45**

**system symbols 51**

## T

**table extension object 80-82**

**table object 74, 75**

access level 199

properties 75-77

**Task Scheduler**

reference link 387

**telemetry 377**

reference link 378

tools, to analyze data 406-408

**Tenant Media Set 252**

**test application**

adding, to existing project 538, 539

**text data**

reading and writing, from blob fields 251

**text selection 33**

**token 622**

**tokenization 622**

**Transact-SQL (T-SQL) 384**

**triggers 489**

**TryFunctions**

used, for error handling 202, 203

## U

**Uniform Resource Identifiers (URIs) 438**

**Universal Code initiative 10**

benefits 11

issues 11

prospected fees timeline 12

references 12

**Universal Print connector**

deploying 330-337

prerequisites 331

**Update AL-Go System Files workflow 535**

**UploadIntoStream method 245**

references 246

## V

**Value Added Resellers (VAR) 14**

**vendor quality implementations 145**

codeunit definition 154-157

page definition 148-151

pageextension definition 151-153

table definition 145-148

**virtual tables**

used, for exposing Dynamics 365 Business  
Central data to Dataverse 595-603

**Visual Studio Code 22**

Activity bar 25

breadcrumbs 35

code block folding 34

code editor 23, 24

comment lines 33

delimiter matching 33

download link 22

editing features 32

Find All References 37

Go to definition 36

IntelliSense 36

mini-map 35

multiple cursors 34

panels area 30, 31

Peek feature 37, 38

sidebar 26

status bar 24

symbols, renaming 38

text selection 33

user interface 22, 23

word completion 36

**Visual Studio Code debugger sections** 351, 352

- debugger sidebar 352
- debugger toolbar 354, 355
- debugging in attach mode 355
- non-debuggable items 356-358

**vNext** 14**W****W1** 3**Waldo** 609

- tools 610, 611

**Waldo GitHub repo** 616**waldo's CRS AL Language Extension** 612, 613

- feedback to Waldo 616
- Renaming / Reorganizing files 613-615
- Run objects 613
- search on Google/Microsoft Docs 615
- snippets 615, 616

**Wave 1** 4**Wave 2** 4**webhooks** 482

- reference link 485
- using 482-485

**word completion** 36**workbooks** 402-405**workspace**

- creating 70

**X****XML document**

- handling, with AL language 214-217

**XmlDocument data type** 256**XML files**

- handling, with AL 256-258

**XMLport object properties**

- reference link 254

**XMLport object triggers**

- reference link 254

**XMLports** 89-91

- using, in AL code 254, 255

**Z****Zebra**

- URL 345

# Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781837630646>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly



