Deepak Agarwal, Abhimanyu Singh

Dynamics 365 for Finance and Operations Development Cookbook

Fourth Edition

Build extensive, powerful, and agile business solutions

Packt>

Dynamics 365 for Finance and Operations Development Cookbook

Fourth Edition

Build extensive, powerful, and agile business solutions

Deepak Agarwal

Abhimanyu Singh

BIRMINGHAM - MUMBAI

Dynamics 365 for Finance and Operations Development Cookbook

Fourth Edition

Copyright © 2017 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: December 2009

Second edition: May 2012

Third edition: April 2015

Fourth edition: August 2017

Production reference: 1100817

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78646-886-4

www.packtpub.com

Credits

Authors
Deepak Agarwal
Abhimanyu Singh

Copy Editor
Safis Editing

Reviewer
Santosh Paruvella

Project Coordinator
Prajakta Naik

Commissioning Editor
Aaron Lazer

Proofreader
Safis Editing

Acquisition Editor
Denim Pinto

Indexer
Francy Puthiry

Content Development Editor
Lawrence Veigas

Graphics
Abhinash Sahu

Technical Editor
Mehul Singh

Production Coordinator
Nilesh Mohite

About the Authors

Deepak Agarwal is a Microsoft Certified Professional who has more than 6 years of relevant experience. He has worked with different versions of Axapta, such as AX 2009, AX 2012, and Dynamics 365. He has had a wide range of development, consulting, and leading roles, while always maintaining a significant role as a business application developer. Although his strengths are rooted in X++ development, he is a highly regarded developer and expert in the technical aspects of Dynamics AX development and customization. He has also worked on base product development with the Microsoft team.

He was awarded the Most Valuable Professional (MVP) award from Microsoft for Dynamics AX four times in a row, and he has held this title since 2013.

He shares his experience with Dynamics AX on his blog: `Axapta V/s Me`

Deepak has also worked on the following Packt books:

1. *Microsoft Dynamics AX 2012 R3 Reporting Cookbook*
2. *Dynamics AX 2012 Reporting Cookbook*
3. *Microsoft Dynamics AX 2012 Programming: Getting Started*

I would like to thank my wife for her support during this duration. Big thanks for her understanding while I spent late hours working on this book. Thanks to my co-author, Abhimanyu, and the Packt team for their support and efforts.

Abhimanyu Singh works as a Microsoft Dynamics 365 for Finance and Operations consultant. Since the start of his career in 2012, he has worked in the development and designing of business solutions for customers in supply chain management, banking, and finance domain using Microsoft technologies. He has several certifications, including the Microsoft Certified Dynamics Specialist certification.

I would like to thank my parents, sister, and brother-in-law for their support and inspiration during the time spent on this book. Secondly, I wish to thank the co-author of this book, and my friend, Deepak Agarwal--a very experienced Dynamics AX consultant.

About the Reviewer

Santosh Paruvella has 12 years of experience in Dynamics AX, and he has worked on various versions of it, from 3.0 to 2012, and Dynamics 365 for Finance and Operations. He is presently working as a Technical Architect and Lead for various implementation projects, designing the solutions and leading the team towards successful implementations.

I have got the chance to review this Dynamics 365 for finance and Operations Development Cookbook, and I am very thankful to the Packt team and the author for this opportunity. This is a very good book for beginners to start with AX development.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com. Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.comand as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.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.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

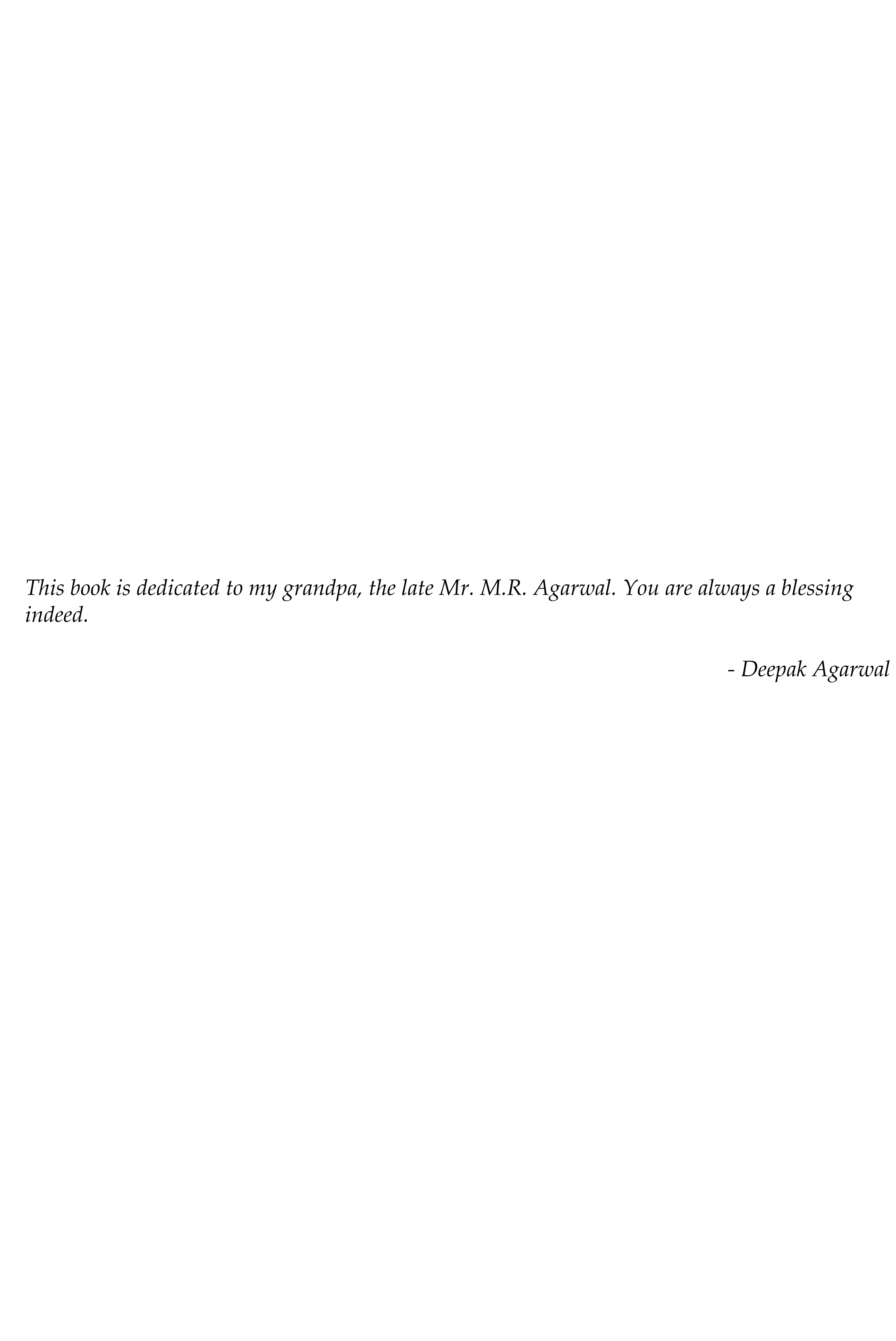
Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1786468867`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!



This book is dedicated to my grandpa, the late Mr. M.R. Agarwal. You are always a blessing indeed.

- Deepak Agarwal

Table of Contents

Preface 1

Chapter 1: Processing Data 8

- **Introduction** 8
- **Creating a new project, package, and model** 9
  - How to do it... 9
  - There's more... 13
- **Creating a new number sequence** 14
  - How to do it... 15
  - How it works... 21
  - See also 21
- **Renaming the primary key** 22
  - How to do it... 25
  - How it works... 28
- **Adding a document handling note** 28
  - Getting ready 29
  - How to do it... 30
  - How it works... 32
- **Using a normal table as a temporary table** 32
  - How to do it... 33
  - How it works... 34
- **Copying a record** 34
  - How to do it... 35
  - How it works... 37
  - There's more... 38
- **Building a query object** 39
  - How to do it... 40
  - How it works... 41
  - There's more... 42
    - Using the OR operator 43
  - See also 44
- **Using a macro in a SQL statement** 44
  - How to do it... 45
  - How it works... 46
- **Executing a direct SQL statement** 47

How to do it... 47
How it works... 50
There's more... 51
Enhancing the data consistency checks 53
Getting ready 53
How to do it... 55
How it works... 57
There's more... 58
Using the date effectiveness feature 58
How to do it... 59
How it works... 62
Chapter 2: Working with Forms 63
Introduction 63
Creating dialogs using the RunBase framework 64
How to do it... 65
How it works... 69
Handling the dialog event 70
How to do it... 71
How it works... 75
See also 76
Creating dialogs using the SysOperation framework 76
Getting ready 77
How to do it... 78
Building a dynamic form 86
How to do it... 87
How it works... 91
Adding a form splitter 93
How to do it... 94
How it works... 95
Creating a modal form 96
How to do it... 96
How it works... 98
There's more... 98
See also 98
Modifying multiple forms dynamically 98
How to do it... 99
How it works... 101
Storing the last form values 101
How to do it... 102

How it works... 104
Using a tree control 105
How to do it... 106
How it works... 111
See also 112
Adding the View details link 112
How to do it... 113
How it works... 115
Selecting a form pattern 116
How to do it 116
Full list of form patterns 117
How to do it... 118
Creating a new form 121
Getting ready 121
How to do it... 121
How it works... 126
Chapter 3: Working with Data in Forms 127
Introduction 127
Using a number sequence handler 128
How to do it... 128
How it works... 131
See also 132
Creating a custom filter control 132
How to do it... 133
How it works... 137
See also 138
Creating a custom instant search filter 138
How to do it... 138
How it works... 140
See also 141
Building a selected/available list 141
How to do it... 142
How it works... 146
There's more... 147
Creating a wizard 151
How to do it... 152
How it works... 162
Processing multiple records 164
How to do it... 164

How it works... 167
Coloring records 168
Getting ready 168
How to do it... 168
How it works... 169
See also 170
Adding an image to records 170
How to do it... 171
How it works... 172
There's more... 172
Displaying an image as part of a form 173
Saving a stored image as a file 175
Chapter 4: Building Lookups 179
Introduction 179
Creating an automatic lookup 180
How to do it... 180
How it works... 181
There's more... 181
Creating a lookup dynamically 183
How to do it... 184
How it works... 186
There's more... 187
Using a form to build a lookup 187
How to do it... 187
How it works... 191
See also 193
Building a tree lookup 193
How to do it... 193
How it works... 196
See also 197
Displaying a list of custom options 197
How to do it... 198
How it works... 200
There's more... 200
Displaying custom options in another way 200
How to do it... 201
How it works... 204
There's more... 204
Building a lookup based on the record description 207

How to do it... 207
How it works... 209
There's more... 210
Building the browse for folder lookup 213
How to do it... 214
How it works... 218
There's more... 218
Creating a color picker lookup 219
How to do it... 220
How it works... 222
Chapter 5: Processing Business Tasks 223
Introduction 223
Using a segmented entry control 224
How to do it... 224
How it works... 227
There's more... 227
See also 229
Creating a general journal 230
How to do it... 230
How it works... 235
There's more 235
See also 238
Posting a general journal 238
How to do it... 238
How it works... 240
See also 240
Processing a project journal 241
How to do it... 241
How it works... 243
There's more... 244
Creating and posting a ledger voucher 245
How to do it... 245
How it works... 248
See also 250
Changing an automatic transaction text 250
Getting ready 251
How to do it... 251
How it works... 253
There's more... 253

Creating a purchase order 254
How to do it... 255
How it works... 256
There's more... 257
Posting a purchase order 257
How to do it... 257
How it works... 259
There's more... 260
Creating a sales order 261
How to do it... 261
How it works... 262
There's more... 263
Posting a sales order 263
How to do it... 264
How it works... 265
There's more... 265
Creating an electronic payment format 266
How to do it... 266
How it works... 271
Chapter 6: Data Management 273
Introduction 273
Data entities 274
Getting ready 274
How to do it... 275
How it works... 280
There's more... 281
Building a data entity with multiple data sources 283
How to do it... 283
How it works... 289
There's more... 290
Data packages 292
Getting ready... 292
How to do it... 294
There's more... 299
See also 302
Data migration 302
Getting ready 303
How to do it... 305
How it works... 308

Importing data 308
How to do it... 308
How it works... 314
Troubleshooting 314
Getting ready 314
How to do it... 315
How it works... 322
There's more... 326
Chapter 7: Integration with Microsoft Office 327
Introduction 327
Configuring and using the Excel Data Connector add-in 328
How to do it... 328
How it works... 332
Using Workbook Designer 333
How to do it... 333
How it works... 336
Export API 336
How to do it... 337
How it works... 339
Lookup in Excel - creating a custom lookup 340
How to do it... 340
How it works... 341
Document management 342
How to do it... 342
How it works... 344
There's more... 344
Chapter 8: Integration with Power BI 346
Introduction 346
Configuring Power BI 347
How to do it... 347
How it works... 354
There's more... 355
See also 355
Consuming data in Excel 356
How to do it... 356
How it works... 361
See also 362
Integrating Excel with Power BI 363

How to do it... 363
How it works... 366
See also 366
Developing interactive dashboards 366
How to do it... 367
How it works... 374
Embedding Power BI visuals 374
How to do it... 374
How it works... 376
Chapter 9: Integration with Services 377
Introduction 377
Authenticating a native client app 378
Getting ready 378
How to do it... 378
How it works... 386
There's more... 387
See also 387
Creating a custom service 388
Getting ready 388
How to do it... 388
How it works... 392
Consuming custom services in JSON 392
Getting ready 393
How to do it... 393
How it works... 396
There's more... 396
Consuming custom services in SOAP 397
Getting ready 397
How to do it... 398
How it works... 400
Consuming OData services 401
Getting ready 401
How to do it... 402
How it works... 405
There's more... 406
See also 406
Consuming external web services 406
Getting ready 406
How to do it... 406

How it works... 412
There's more... 413
See also 414
Chapter 10: Improving Development Efficiency and Performance 415
Introduction 415
Using extensions 416
How to do it... 416
How it works... 419
Caching a display method 420
How to do it... 420
How it works... 422
There's more... 423
Calculating code execution time 424
How to do it... 424
How it works... 425
There's more... 426
Enhancing insert, update, and delete operations 427
How to do it... 427
How it works... 434
There's more... 435
Using delete_from 435
Using update_recordSet for faster updates 436
Writing efficient SQL statements 437
How to do it... 437
How it works... 439
There's more... 440
See also 441
Using event handler 441
How to do it... 442
How it works... 444
There's more... 444
Creating a Delegate method 445
Getting ready... 445
How to do it... 445
How it works... 447
There's more... 447
See also 447
Index 448

Preface

As a Dynamics 365 for Finance and Operations developer, your responsibility is to deliver all kinds of application customization, whether small adjustments or a bespoke modules. Dynamics 365 for Finance and Operations is a highly customizable system and requires a significant amount of knowledge and experience to deliver quality solutions. One goal can be achieved in multiple ways, and there is always the question of which way is the best.

This book takes you through numerous recipes to help you with daily development tasks.

Each recipe contains detailed step-by-step instructions along with the application screenshots and in-depth explanations. The recipes cover multiple Dynamics 365 for Financial and Operations modules, so, at the same time, the book provides an overview of the functional aspects of the system for developers.

What this book covers

`Chapter 1`, *Processing Data*, focuses on data manipulation. It explains how to build data queries, check and modify existing data, read and write external files, and use data effectively.

`Chapter 2`, *Working with Forms*, covers various aspects of building forms in Dynamics 365 for Finance and Operations. In this chapter, dialogs and their events are explained. Also, various useful features, such as splitters, tree controls, and checklists, are explained.

`Chapter 3`, *Working with Data in Forms*, basically supplements *Chapter 2*, *Working with Forms*, and explains the data organization in forms. The examples in this chapter include instructions for building filter controls on forms, processing multiple records, and working with images and colors.

`Chapter 4`, *Building Lookups*, covers all kinds of lookups in the system. This chapter starts with a simple, automatically generated lookup, continues with more advanced ones, and finishes with standard Windows lookups, such as the file selection dialog and the color picker.

`Chapter 5`, *Processing Business Tasks*, explains how to use the Dynamics 365 for Finance and Operations business logic API. In this chapter, topics such as how to process journals, purchase orders, and sales orders are discussed. Other features, such as posting ledger vouchers, modifying transaction texts, and creating electronic payment formats, are included as well.

`Chapter 6`, *Data Management*, explains the data management and data entity concepts, how to build a data entity, data packages, and import and export in Dynamics 365 for Financial and Operations.

`Chapter 7`, *Integration with Microsoft Office*, explains how to configure and use the Excel Data Connector add-in, and design Excel workbooks with the data feed from Dynamics 365 from Operations using OData. It also covers how to use the export API and document management.

`Chapter 8`, *Integration with Power BI*, explains the configuration of Power BI and its integration with Dynamics 365 for Financial and Operations to develop interactive dashboards and embed them in Dynamics 365 for Financial and Operations workspaces.

`Chapter 9`, *Integration with Services*, explains how to use services in Dynamics 365 for Financial and Operations. This chapter covers how to create services, authentication, SOAP applications, JSON applications, and OData services. It also demonstrates how to consume external services.

`Chapter 10`, *Improving Development Efficiency and Performance*, presents a few ideas on how to make daily development tasks easier. It discusses how system performance can be improved by following several simple rules. This chapter explains how to calculate code execution time, how to write efficient SQL statements, and how to properly cache display methods.

Exceptions and considerations

The code in this book follows the best practice guidelines provided by Microsoft, but there are some exceptions:

- No text labels were used to make the code clear
- No three-letter code was used in front of each new AOT object
- No configuration or security keys were used
- Object properties that are not relevant to the topic being discussed are not set

Also, here are some considerations that you need to keep in mind when reading this book:

- Each recipe only demonstrates the principle and is not a complete solution
- The data in your environment might not match the data used in the recipes, so the code might have to be adjusted appropriately
- For each recipe, the assumption is that no other modifications are present in the system, unless it is explicitly specified

- The code might not have all the possible validations that are not relevant to the principle being explained
- The code might have more variables than required in order to ensure that it is clear for all audiences
- Sometimes, unnecessary code wrapping is used to make sure the code fits into the page width of this book and is easily readable

What you need for this book

All the coding examples were performed in a Microsoft Azure-hosted Microsoft Dynamics 365 for Financial and Operations environment. The following list of software from the virtual image was used in this book:

- Microsoft Dynamics 365 for Financial and Operations (Update 6)
- Microsoft Visual studio 2015
- Microsoft Windows Server 2015 Enterprise
- Microsoft SQL Server 2016
- Microsoft Power BI
- Microsoft Office Excel 2015
- Microsoft Office Word 2015
- Microsoft Internet Explorer
- Windows Notepad

Although all the recipes have been tested on the previously-mentioned software, they may work on older or newer software versions with minor code adjustments. As Microsoft is continuously evolving on Dynamics 365 for Financial and Operations, we might see some differences while using the same code in older or newer updates of application. Stick to the concept and customize or extend the application.

Who this book is for

If you are a Dynamics AX developer primarily focused on delivering time-proven applications, then this book is for you. This book is also ideal for people who want to raise their programming skills above the beginner level, and, at the same time, learn the functional aspects of Dynamics 365 for Financial and Operations. Some X++ coding experience is expected.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Then, to override the data source's `write()` method."

A block of code is set as follows:

```
[FormDataSourceEventHandler(formDataSourceStr(CustGroup,
 CustGroup), FormDataSourceEventType::Written)]
public void CustGroup_OnWritten(FormDataSource sender,
FormDataSourceEventArgs e)
{
  this.numberSeqFormHandler().formMethodDataSourceWrite();
}
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Dynamics-365-for-Finance-and-Operations-Development-Cookbook-Fourth-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to
`https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at
`questions@packtpub.com`, and we will do our best to address the problem.

1
Processing Data

In this chapter, we will cover the following recipes:

- Creating a new project, package, and model
- Creating a new number sequence
- Renaming the primary key
- Adding a document handling note
- Using a normal table as a temporary table
- Copying a record
- Building a query object
- Using a macro in a SQL statement
- Executing a direct SQL statement
- Enhancing the data consistency checks
- Using the date effectiveness feature

Introduction

This chapter focuses on data manipulation exercises in all new Dynamics 365 for Finance and Operations. These exercises are very useful when doing data migration, system integration, custom reporting, and so on. Here, we will discuss how to work with query objects from the X++/C# code. We will also discuss how to reuse macros in X++ SQL statements and how to execute SQL statements directly to the database. This chapter will explain how to rename primary keys, how to merge and copy records, how to add document handling notes to selected records, and how to create and read XML and comma-separated files. The chapter ends with a recipe about the date effectiveness feature.

Creating a new project, package, and model

Elements in Dynamics 365 for Finance and Operations represent every individual element of AOT such as class, table, form, and so on. Elements in Dynamics 365 for Finance and Operations are stored on disk as XML files; these files contain the metadata and source code for the element. The XML files are the unit of Source Control.

Projects works the same as AX2012, but in D365 an element can be customized only once they are added to a specific Visual Studio project. The project may only belong to one model.

A **Dynamics 365 for Finance and Operations model** is a group of elements. Standard elements are part of a standard model; you can add them into your model and do customization. A model is a design-time concept. An example of models: warehouse management model, a project accounting model, and more. Models can have one or more projects. Models may only belong to one package.

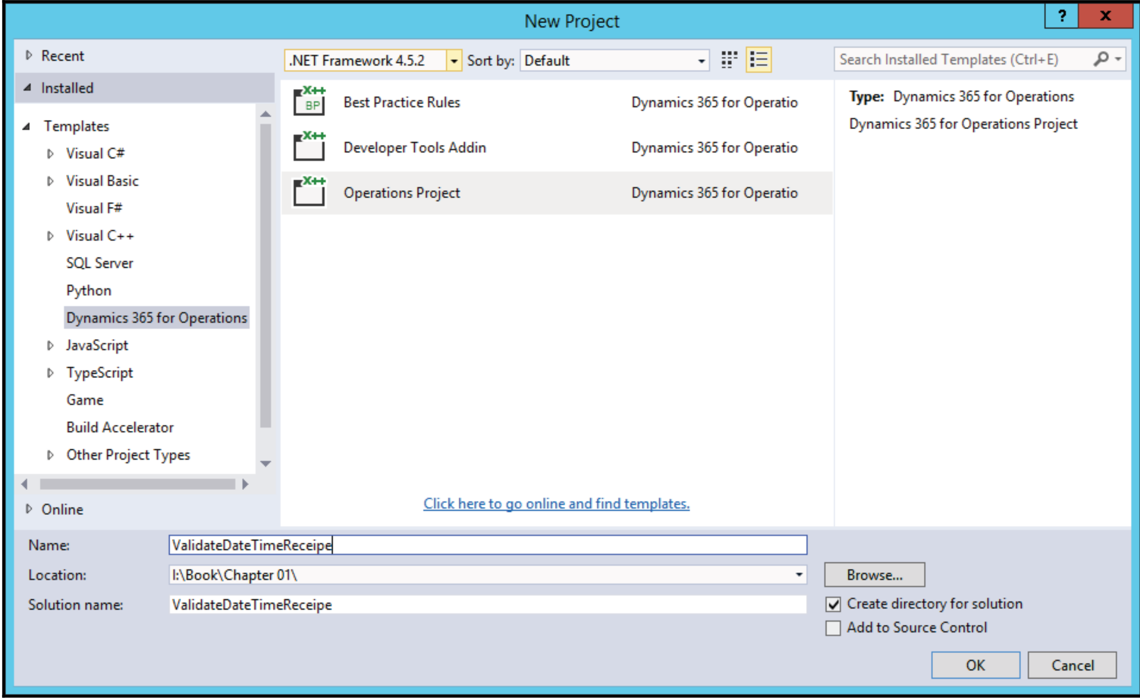
A **Dynamics 365 for Finance and Operations package** is a deployment and compilation unit of one or more models. It includes model metadata, binaries, cubes, and other associated resources. One or more D365 packages can be packaged into a deployment package, which is the vehicle used for deployment on UAT and production environments. Packages are packaged into a deployable package file for deployment to Sandbox or production environments. A package can have one or more models. Packages can have references to other packages, just like .NET assemblies can reference each other.

How to do it...

To create a new project, follow these steps:

1. Open Visual Studio as admin.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the list of template types, expand the **Installed** node.
4. Expand the **Templates** node.
5. Select the **Microsoft Dynamics 365 for Operations** category.
6. Select the **D365 Project** template.
7. Enter the name and location for the new project.

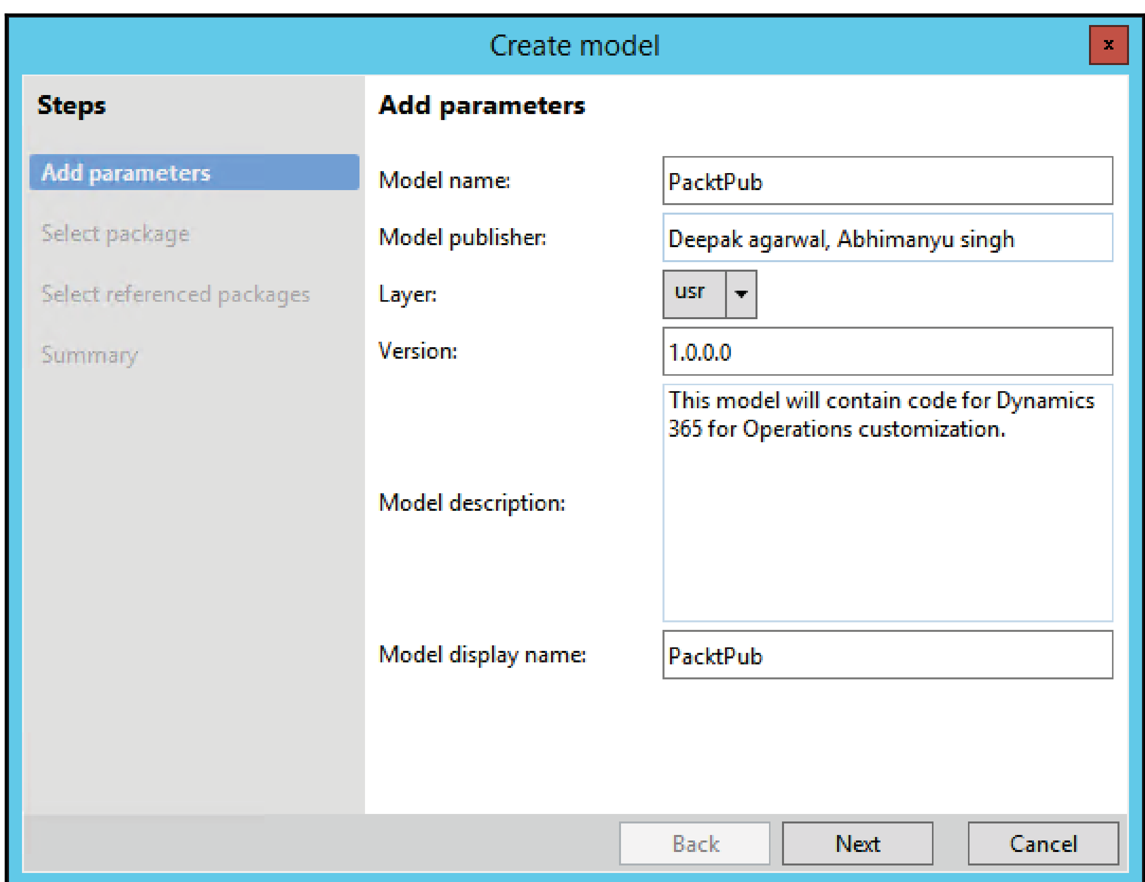
8. Select **Create directory for solution** if you want to create a new solution for this project, uncheck if you want to add in your current solution.



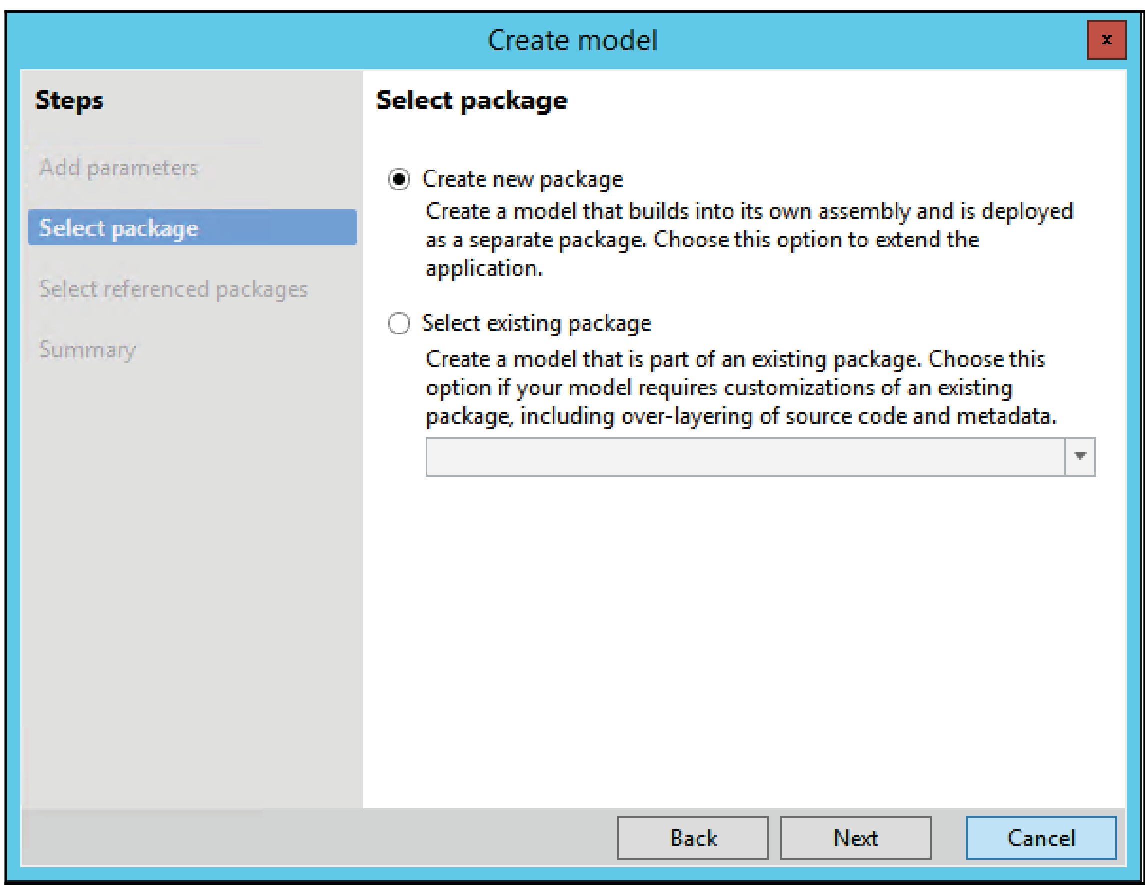
To create a new model, follow these steps:

1. Open Visual Studio as admin.
2. On the **Dynamics 365** menu, point to **Model management** and select **Create model**.

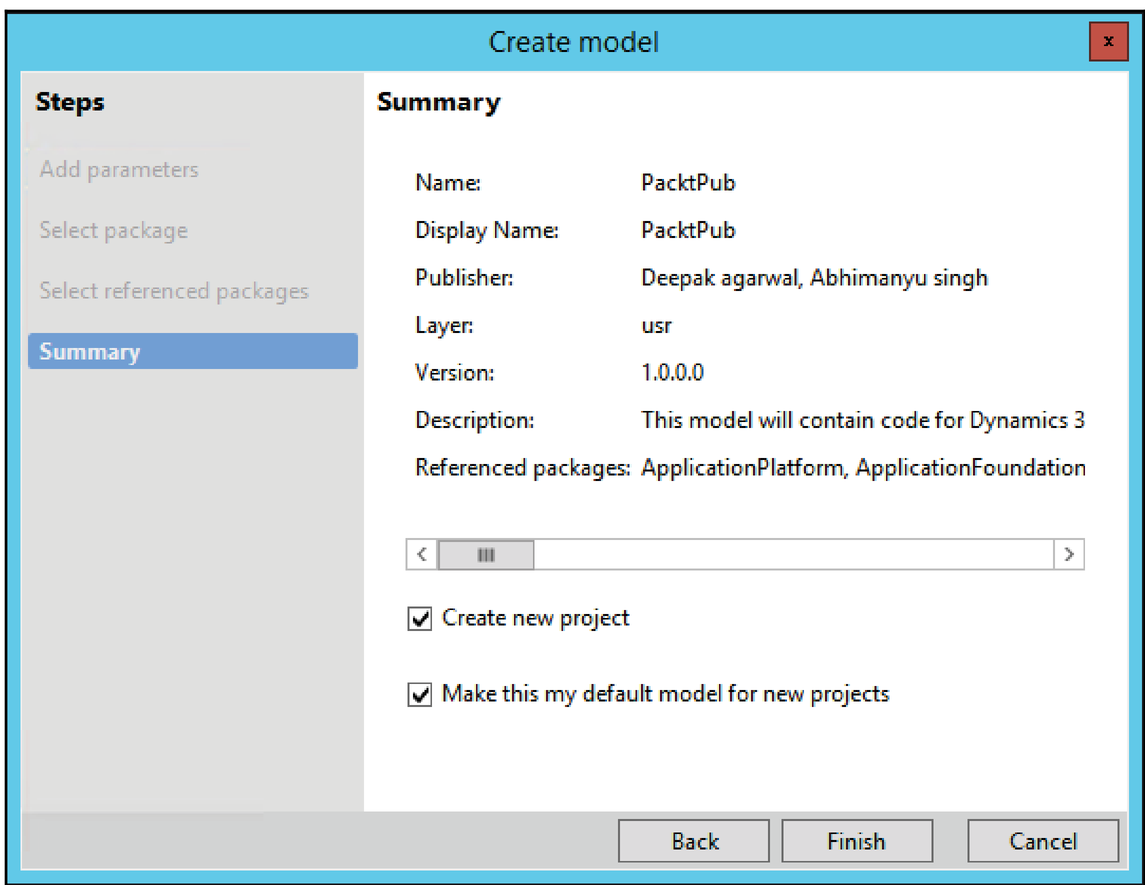
3. Give a model, publisher name, and other values:



4. Now here you can create a new package or select any existing package. We could create a new package and select the required package as referenced packages:



5. Double-check the summary with details. Select **Create new project** if you want to create a new project in this model once created. You can mark this model to all your new projects by selecting options:



There's more...

As you saw, there was one more step while creating a model, **Select referenced packages**. When you create your own package you can select from an existing package to add them as references in your new package. You may need to add some standard package reference if you want to add them into your customization.

Here are the steps to create a new package:

1. Open Visual Studio as admin.
2. On the **Dynamics 365** menu, point to Model management and select **Create model.**
3. Give a model, publisher name, and other values.
4. On the next step select **Create new package**
5. Give a name to your package.
6. Next select the existing package as a reference to this new package.
7. Click on **Finish**.

So now you have your own model with a new package.

Creating a new number sequence

Number sequences in A Dynamics 365 for Finance and Operations is a deployment and compilation unit of one or more models. It includes model metadata, binaries, cubes, and other associated resources. One or more D365 packages can be packaged into a deployment package, which is the vehicle used for deployment on UAT and production environments. Packages are packaged into a deployable package file for deployment to Sandbox or production environments. A package can have one or more models. Packages can have references to other packages, just like .NET assemblies can reference each other. are used to generate specifically formatted numbers for record identification. These number sequences can be anything from voucher numbers or transaction identification numbers to customer or vendor accounts.

When developing custom functionality, often one of the tasks is to add a new number sequence to the system in order to support newly created tables. Adding a number sequence to the system is a two-step process. First, we create the number sequence itself; second, we start using it in some particular form or from the code.

D365 contains a list of `NumberSeqApplicationModule` derivative classes, which hold the number sequence's setup data for the specific module. These classes are read by the number sequence wizard, which detects existing number sequences and proposes to create the missing ones or newly added ones. The wizard is normally run as a part of the application initialization. It can also be rerun any time later when expanding the D365 functionality used, where a setup of additional number sequences is required. The wizard also has to be rerun if new custom number sequences are added to the system.

In this recipe, we will do the first step, that is, add a new number sequence to the system. In a standard application, the customer group number is not driven by any number sequence, so we will enhance this by creating it. The second step is explained later in the *Using a number sequence handler* recipe in `Chapter 3`, *Working with Data in Forms*.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Create a new `NumberSeqModuleCustomer_packt` class in the D365 Project that extends the `NumberSeqModuleCustomer` class in the **Application** and add the following code snippet at the bottom of the `loadModule_Extension()` method:

```
 class NumberSeqModuleCustomer_packt extends
  NumberSeqModuleCustomer
{
  public void loadModule_Extension()
 {
   NumberSeqDatatype datatype = NumberSeqDatatype::construct();

   datatype.parmDatatypeId(extendedTypeNum(CustGroupId));
   datatype.parmReferenceHelp("Customer group ID");
   datatype.parmWizardIsContinuous(false);
   datatype.parmWizardIsManual(NoYes::No);
   datatype.parmWizardIsChangeDownAllowed(NoYes::Yes);
   datatype.parmWizardIsChangeUpAllowed(NoYes::Yes);
   datatype.parmWizardHighest(999);
   datatype.parmSortField(20);
   datatype.addParameterType(
   NumberSeqParameterType::DataArea, true, false);

   this.create(datatype);
 }
}
```

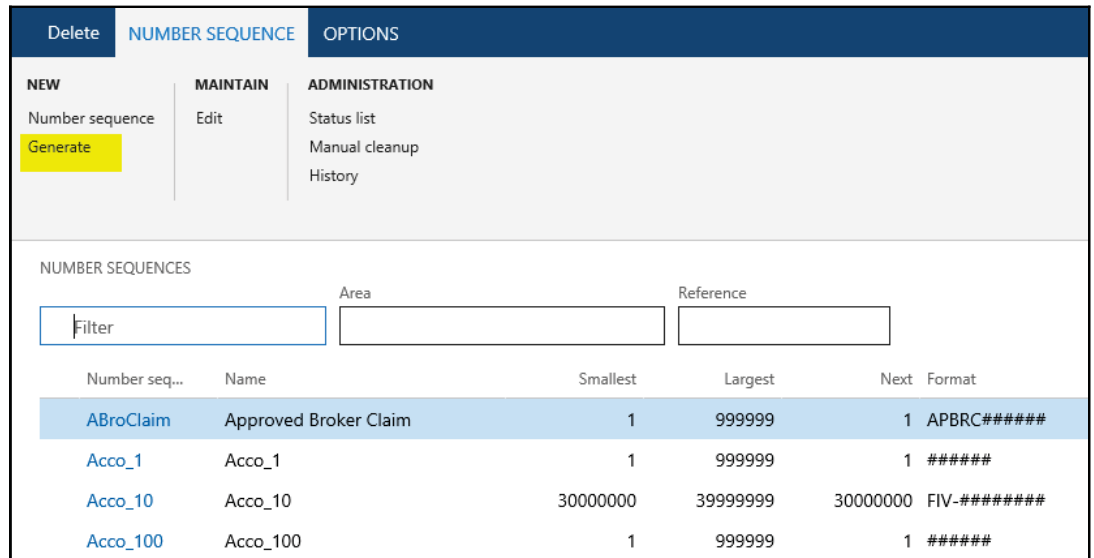
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

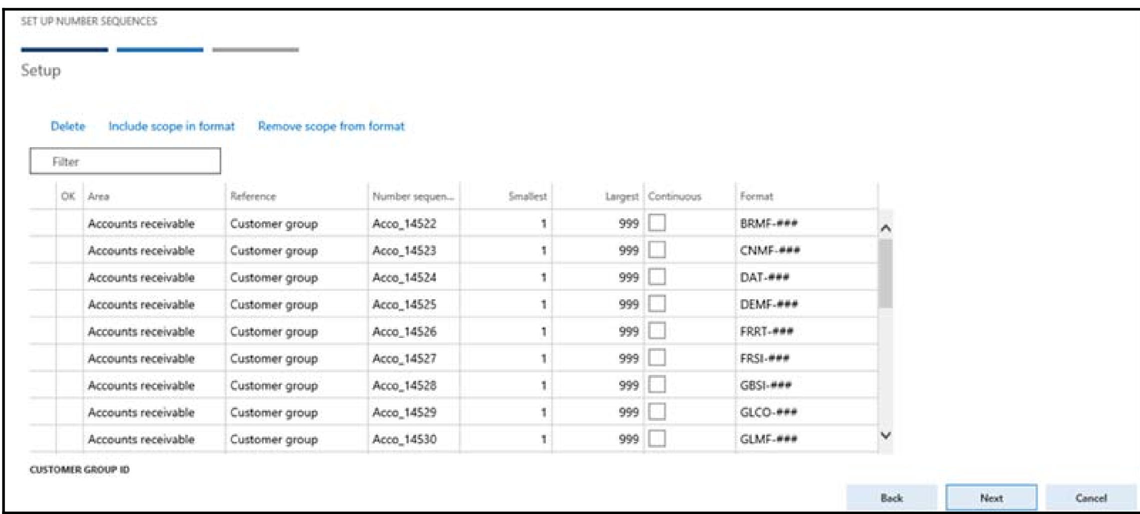
2. Create a new runnable class (Job) with the following lines of code, build the solution and run it:

```
class loadNumSeqCustPackt
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
   public static void Main(Args args)
  {
    //define the class variable
    NumberSeqModuleCustomer_packt nymberSeqMod = new
     NumberSeqModuleCustomer_packt();

    //load the number sequences
     nymberSeqMod.loadModule_Extension();
  }

}
```

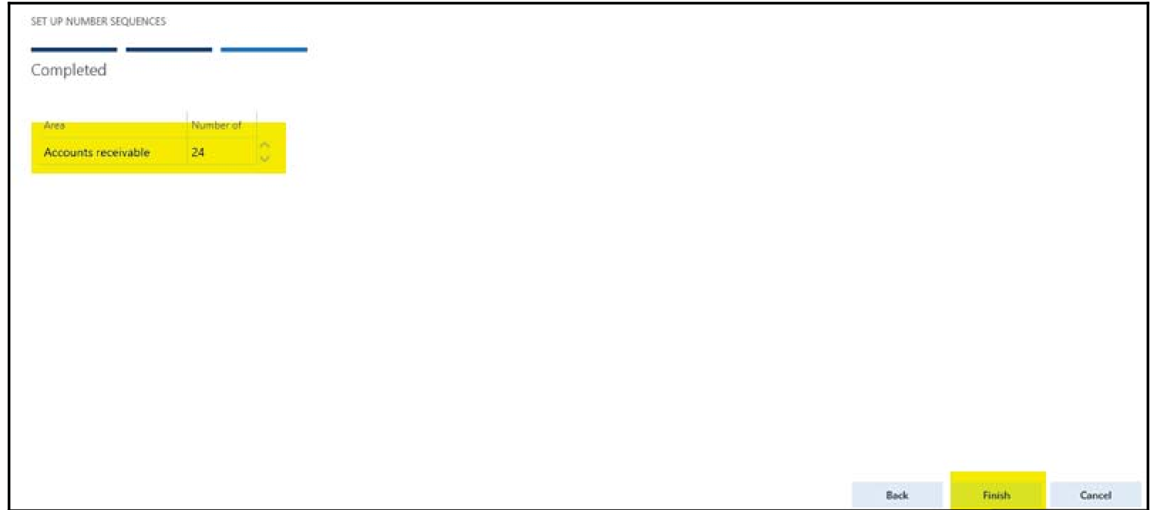
3. Run the number sequence wizard by clicking on the **Generate** button under **Number sequence** by going to **Organization administration** | **Common** | **Number sequence** and then click on the **Next** button, as shown in the following screenshot:



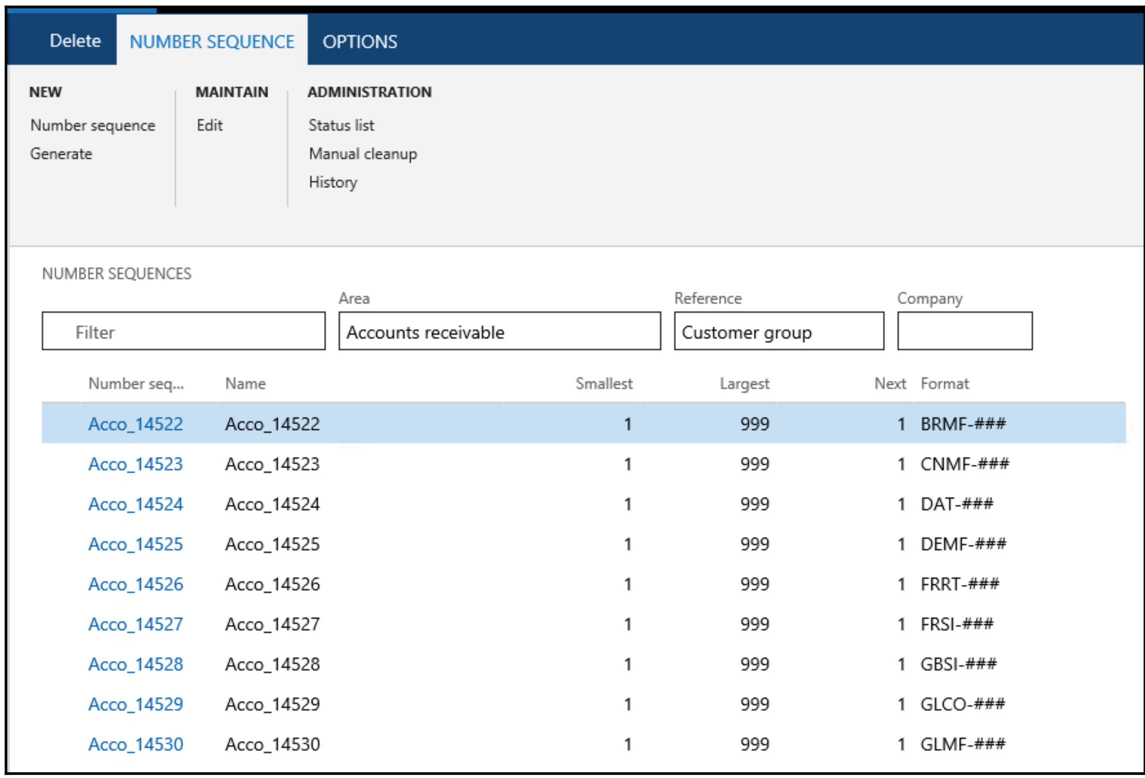
4. Click on **Details** to view more information. Delete everything apart from the rows where **Area** is **Accounts receivable** and **Reference** is **Customer group**. Note the number sequence codes and click on the **Next** button, as shown here:



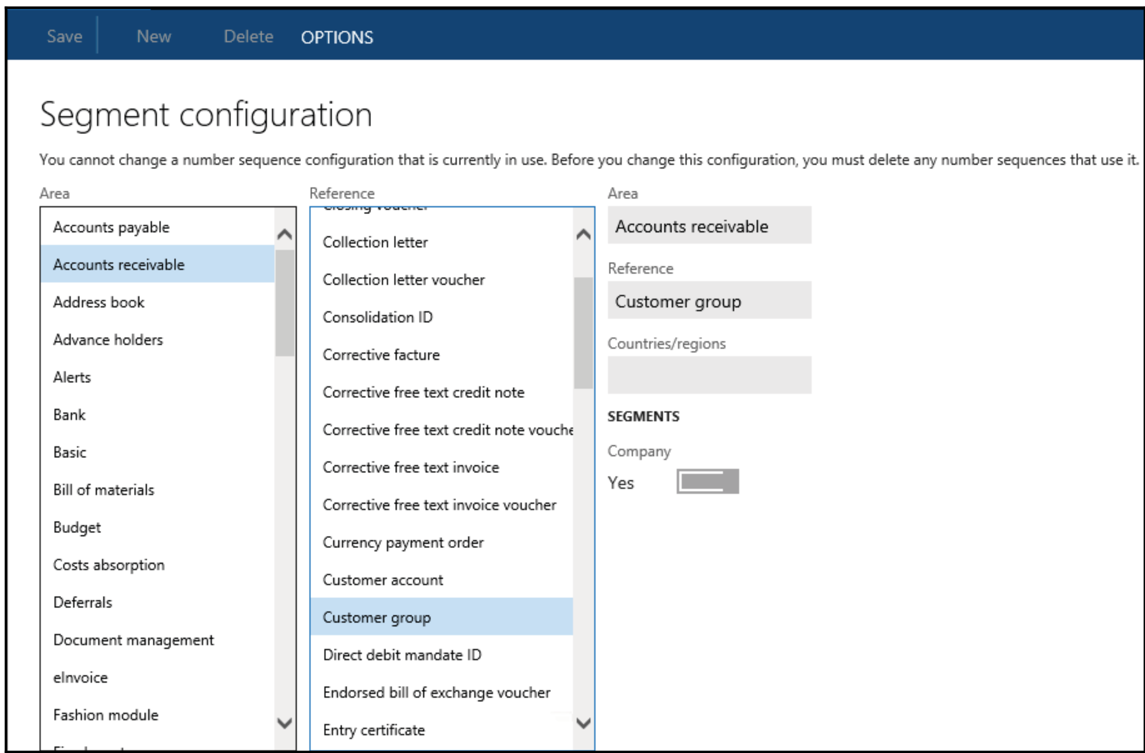
5. On the last page, click on the **Finish** button to complete the setup, as shown in the following screenshot:



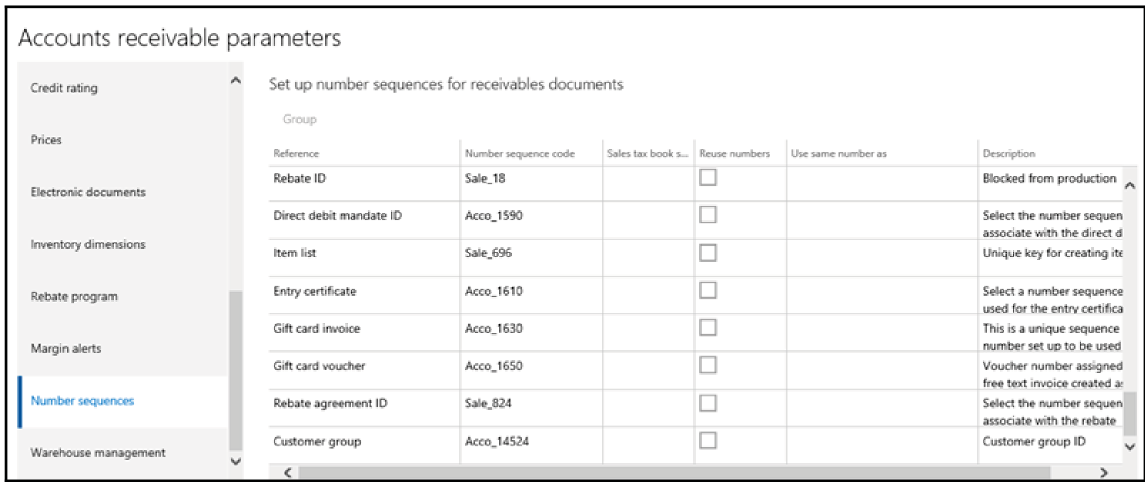
6. The newly created number sequences now can be found in the **Number sequence** form, as shown in the following screenshot:



7. Navigate to **Organization administration** | **Number sequences** | **Segment configuration** and notice the new **Customer group** reference under the **Accounts receivable** area:



8. Navigate to **Accounts receivable** | **Setup** | **Accounts receivable parameters** and select the **Number sequences** tab. Here, you should see the new number sequence code:



9. The last thing to be done is to create a helper method for this number sequence. Create a new extension class `CustParameters_Extension` for the `CustParameters` table and add it to the Dynamics 365 Project and then create the following method and build the solution:

```
[ExtensionOf(tableStr(CustParameters))]
final  class CustParameters_Extension
{
  /// <summary>
  /// Gets the number reference customer group id.
  /// </summary>
  /// <returns>
  /// An instance of the <c>NumberSequenceReference</c> class.
  /// </returns>
  client server static NumberSequenceReference
   numRefCustGroupId()
  {
    NumberSequenceReference NumberSeqReference;
    NumberSeqReference = NumberSeqReference::findReference
    (extendedTypeNum(CustGroupId));
     return NumberSeqReference;
  }
}
```

How it works...

We start the recipe by adding a number sequence initialization code into the `NumberSeqModuleCustomer_packt` class. As understood from its name, the number sequence initialization code holds the initialization of the number sequences that belong to the **Accounts receivable** module.

The code in the `loadModule_Extension()` method defines the default number sequence settings to be used in the wizard, such as the data type, description, and highest possible number. Additional options such as the starting sequence number, number format, and others can also be added here. All the mentioned options can be changed while running the wizard. The `addParameterType()` method is used to define the number sequence scope. In the example, we created a separate sequence for each Legal entity.

Before we start the wizard, we initialize number sequence references. This should be done as a part of the Dynamics 365 for Finance and Operations initialization checklist, but in this example, we execute it manually by calling the `loadModule_Extension()` method of the `NumberSeqApplicationModule_packt` class.

Next, we execute the wizard that will create the number sequences for us. We skip the welcome page and in the second step of the wizard, the **Details** button can be used to display more options. The options can also be changed later in the **Number sequences** form before or even after the number sequence is actually used. The last page shows an overview of what will be created. Once completed, the wizard creates new records in the **Number sequences** form for each company.

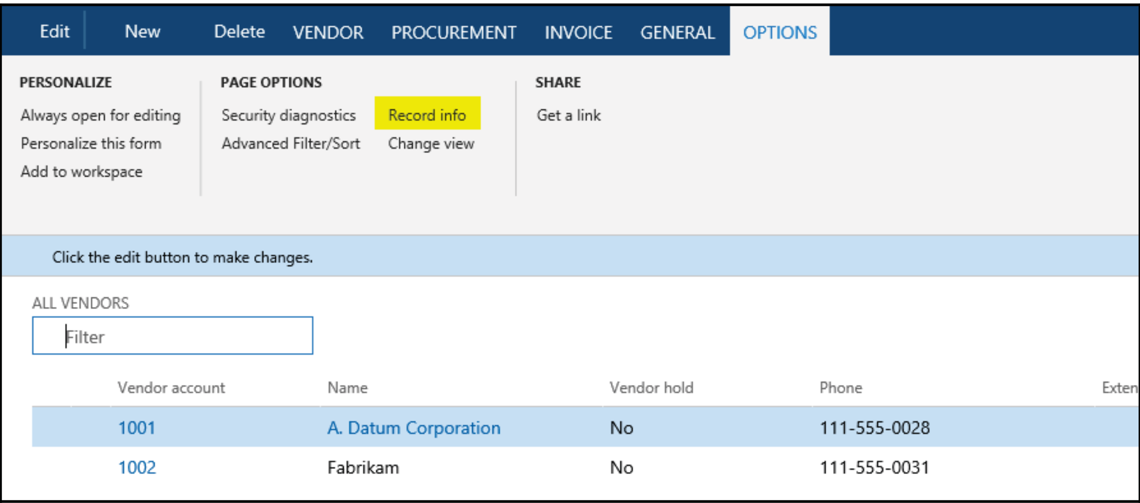
The newly created number sequence reference appears in the **Segment configuration** form. Here, we can see that the **Data area** checkbox is checked, which means that we will have separate number lists for each company. The number sequence setup can be normally located in the module parameter forms.

See also

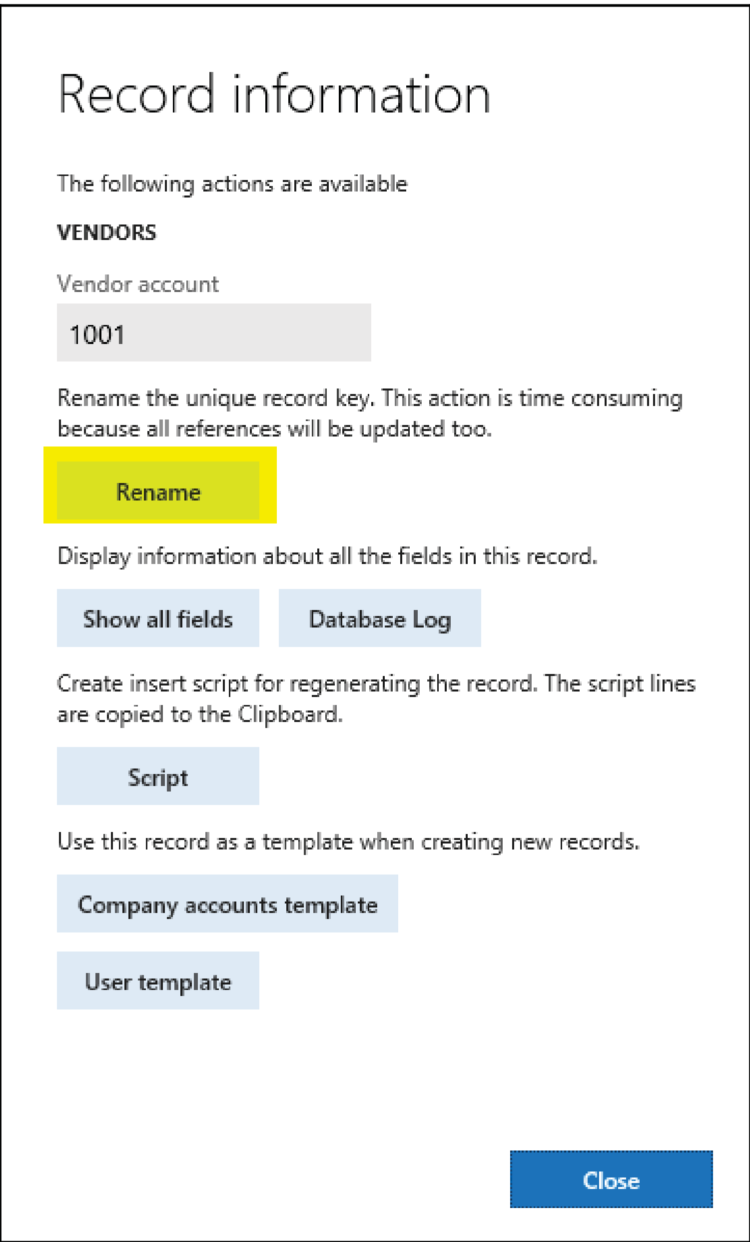
- The *Using a number sequence handler* recipe in `Chapter 3`, *Working with Data in Forms*

Renaming the primary key

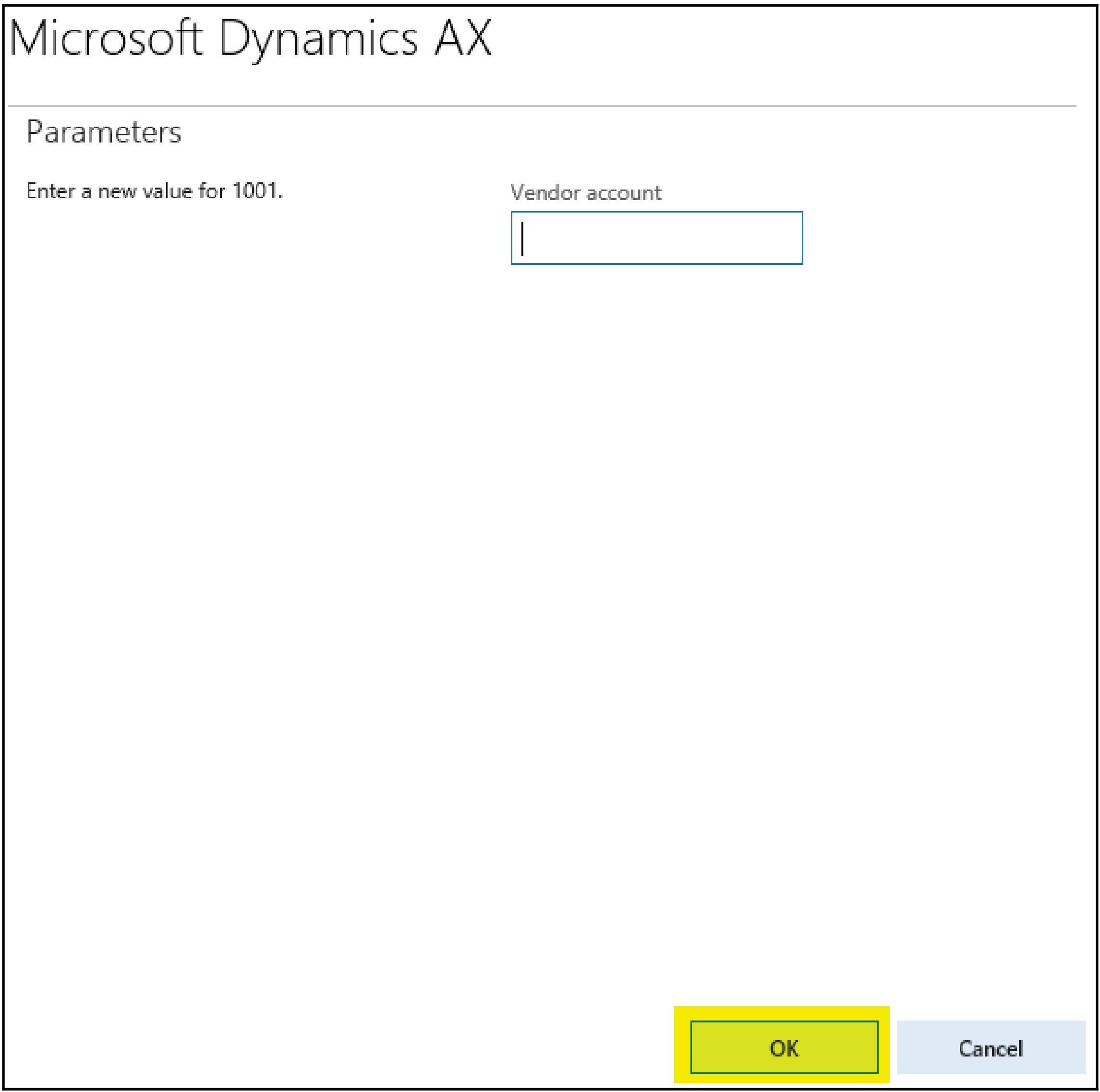
Most of you who are familiar with the Dynamics 365 for Finance and Operations application, have probably used the standard `Rename` function. This function allows you to rename the primary key of almost any record. With this function, you can fix records that were saved or created by mistake. This function ensures data consistency, that is, all the related records are renamed as well. It can be accessed from the **Record information** form (shown in the following screenshot), which can be opened by selecting **Record info** from the right-click menu on any record:



A new form will open as follows:



Click on the **Rename** button to rename the **Vendor Account** field value.



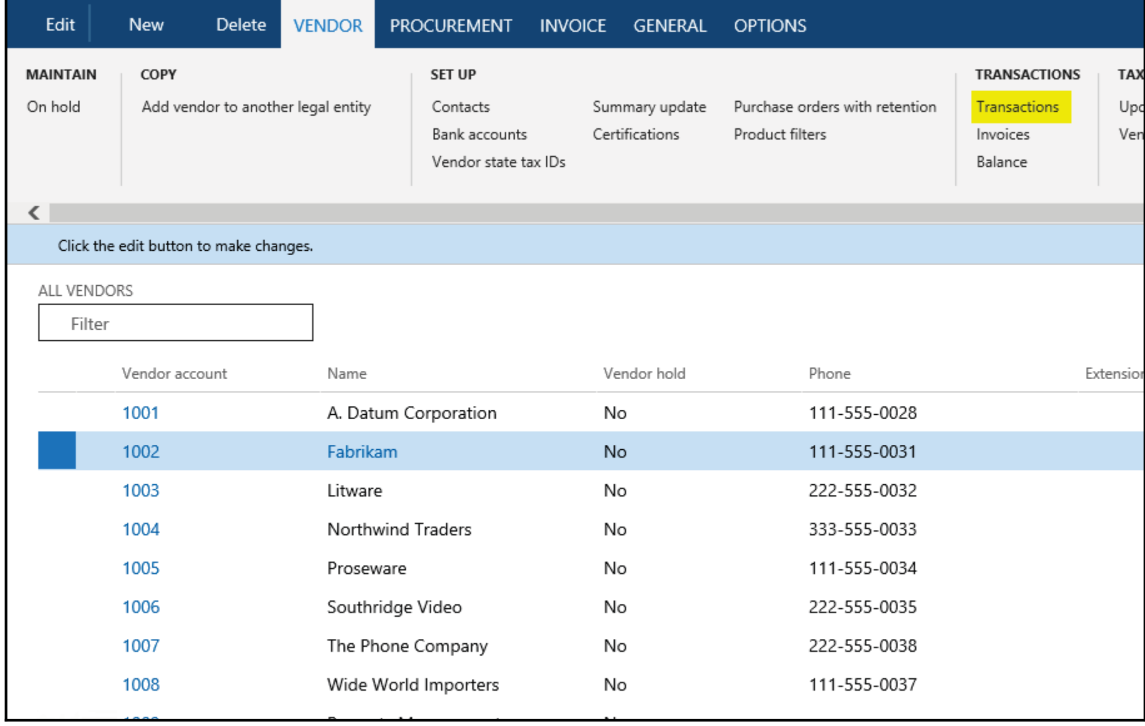
When it comes to mass renaming, this function might be very time-consuming as you need to run it on every record. An alternative way of doing this is to create a job that automatically runs through all the required records and calls this function automatically.

This recipe will explain how the record's primary key can be renamed through the code. As an example, we will create a job that renames a vendor account.

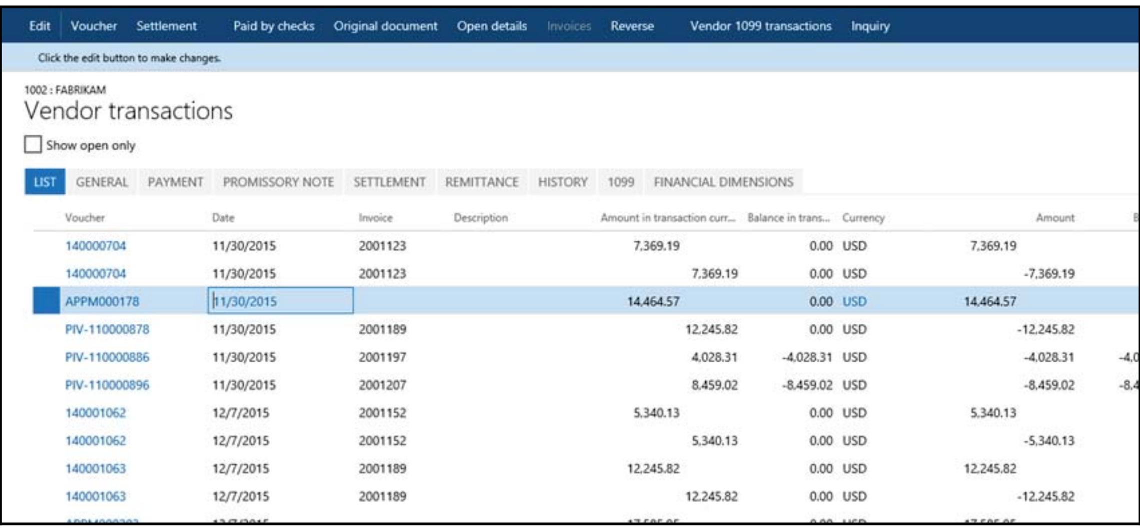
How to do it...

Carry out the following steps in order to complete this recipe:

1. Navigate to **Accounts payable** | **Vendors** | **All vendors** and find the account that has to be renamed, as shown in the following screenshot:



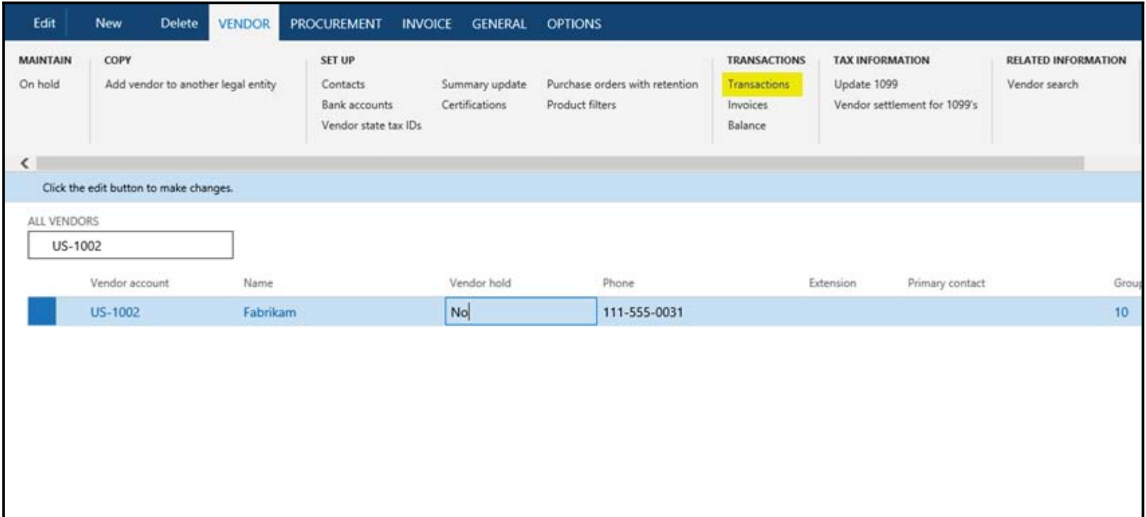
2. Click on **Transactions** in the **Action** pane to check the existing transactions, as shown in the following screenshot:



3. Create a new project, create a runnable class named `VendAccountRename`, and enter the following code snippet. Use the previously selected account:

```
 class VendAccountRename
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
 {
   VendTable vendTable;

   ttsBegin;

   select firstOnly vendTable
   where vendTable.AccountNum == '1002';

   if (vendTable)
  {
    vendTable.AccountNum = 'US-1002';
    vendTable.renamePrimaryKey();
  }

   ttsCommit;
 }

}
```

4. Select class `VendAccountRename` and right-click and then select **Set as startup object**. Execute the class by clicking **Start** in Visual Studio and check whether the renaming was successful, by navigating to **Accounts payable** | **Vendors** | **All vendors** again and finding the new account. The new account should have retained all its transactions and other related records, as shown in the following screenshot:



5. Click on **Transactions** in the **Action** pane in order to see whether the existing transactions are still in place, as shown in the following screenshot:

Edit | Voucher | Settlement | Paid by checks | Original document | Open details | Invoices | Reverse | Vendor 1099 transactions | Inquiry

Click the edit button to make changes.

US-1002 : FABRIKAM

Vendor transactions

Show open only

LIST | GENERAL | PAYMENT | PROMISSORY NOTE | SETTLEMENT | REMITTANCE | HISTORY | 1099 | FINANCIAL DIMENSIONS

Voucher	Date	Invoice	Description		Amount in transaction curr...	Balance in trans...	Currency		Amount
140000704	11/30/2015	2001123		7,369.19		0.00	USD	7,369.19	
140000704	11/30/2015	2001123			7,369.19	0.00	USD		-7,369.19
APPM000178	11/30/2015			14,464.57		0.00	USD	14,464.57	
PIV-110000878	11/30/2015	2001189			12,245.82	0.00	USD		-12,245.82
PIV-110000886	11/30/2015	2001197			4,028.31	-4,028.31	USD		-4,028.31
PIV-110000896	11/30/2015	2001207			8,459.02	-8,459.02	USD		-8,459.02
140001062	12/7/2015	2001152		5,340.13		0.00	USD	5,340.13	
140001062	12/7/2015	2001152			5,340.13	0.00	USD		-5,340.13
140001063	12/7/2015	2001189		12,245.82		0.00	USD	12,245.82	
140001063	12/7/2015	2001189			12,245.82	0.00	USD		-12,245.82

How it works...

In this recipe, we first select the desired vendor record and set its account number to the new value. Note that only the fields belonging to the table's primary key can be renamed in this way.

Then, we call the table's `renamePrimaryKey()` method, which does the actual renaming. The method finds all the related records for the selected vendor account and updates them with the new value. The operation might take a while, depending on the volume of data, as the system has to update multiple records located in multiple tables.

Adding a document handling note

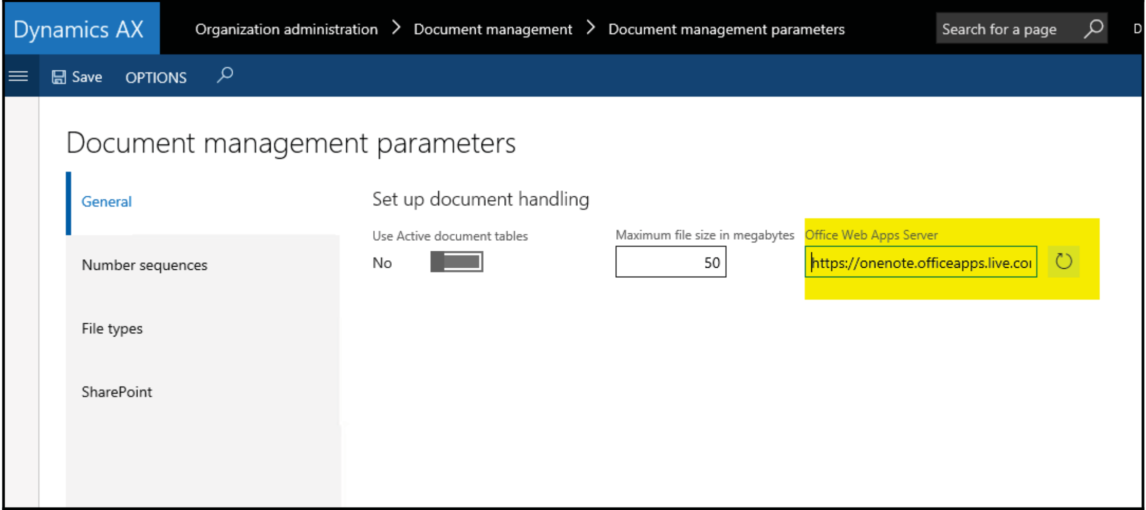
Document handling in Dynamics 365 for Finance and Operations is a feature that allows you to add notes, links, documents, images, files, and other related information to almost any record in the system. For example, we can track all the correspondence sent out to our customers by attaching the documents to their records in Dynamics 365 for Finance and Operations. Document handling on most of the forms can be accessed either from the **Action** pane by clicking on the **Attachments** button and selecting **Document handling** from the **Command** menu under **File** or selecting the **Document handling** icon from the status bar.

Document handling has a number of configuration parameters that you can find by navigating to **Organization administration** | **Setup** | **Document management**. Please refer to Dynamics 365 for Operations Manuals to find out more.

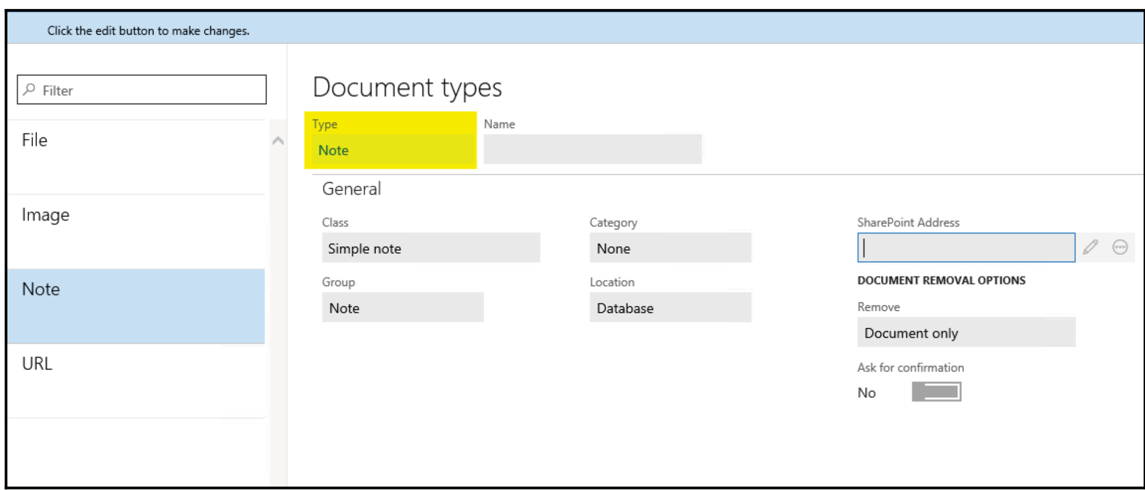
Dynamics 365 for Finance and Operations also allows you to add document handling notes from the code. This can come in handy when you need to automate the document handling process. In this recipe, we will demonstrate this by adding a note to a vendor account.

Getting ready

Before you start, ensure that document handling is enabled on the user interface. Open **Document management parameters** by navigating to **Organization administration** | **Setup** | **Document management** and make sure that **Use Active document tables** is not marked, as shown in the following screenshot:



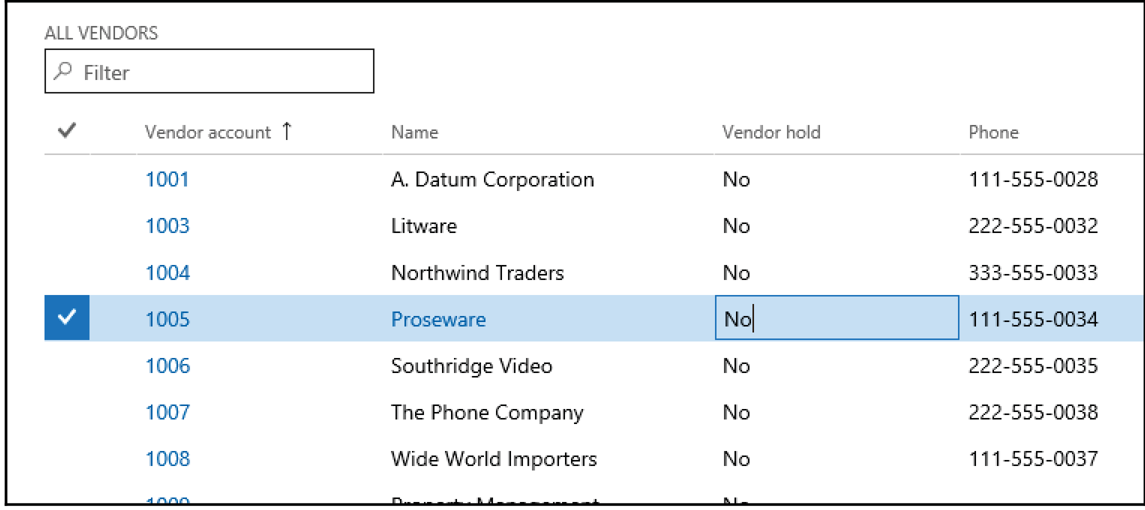
Then, open the **Document types** form from the same location and pick or create a new document type with its **Group** set to **Note**, as shown in the following screenshot. In our demonstration, we will use **Note**.



How to do it...

Carry out the following steps in order to complete this recipe:

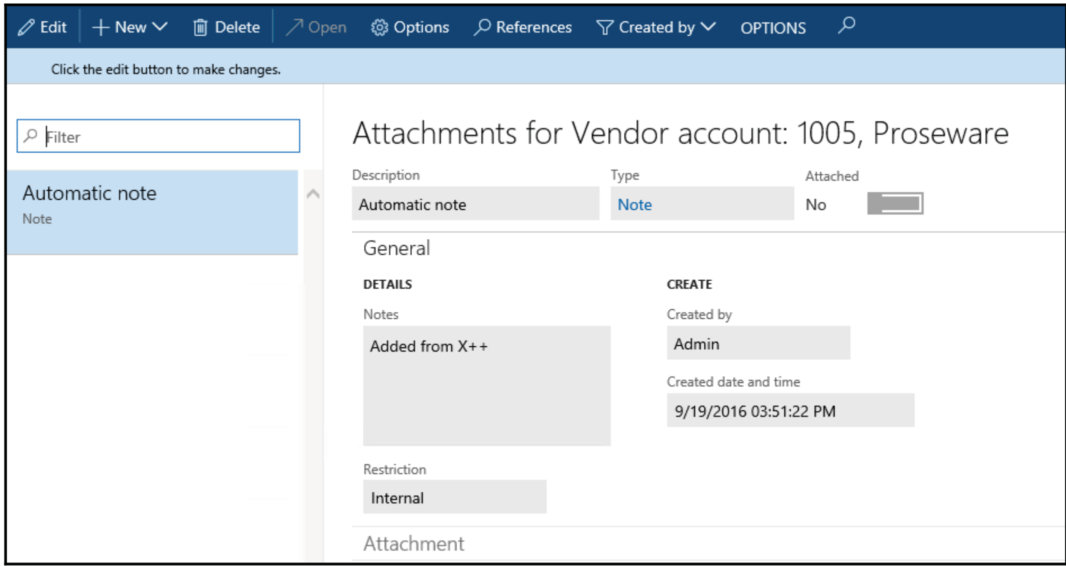
1. Navigate to **Accounts payable** | **Vendors** | **All vendors** and locate any vendor account to be updated, as shown in the following screenshot:



2. Create a Dynamics 365 for Operations Project, create a new runnable class named `VendAccountDocument`, and enter the following code snippet. Use the previously selected vendor account and document type:

```
class VendAccountDocument
{
  static void main(Args _args)
  {
    VendTable vendTable;
    DocuType  docuType;
    DocuRef   docuRef;

    vendTable = VendTable::find('1005');
    docuType  = DocuType::find('Note');

    if (!docuType ||
     docuType.TypeGroup != DocuTypeGroup::Note)
   {
     throw error("Invalid document type");
   }


    docuRef.RefCompanyId = vendTable.dataAreaId;
    docuRef.RefTableId   = vendTable.TableId;
    docuRef.RefRecId     = vendTable.RecId;
    docuRef.TypeId       = docuType.TypeId;
    docuRef.Name         = 'Automatic note';
    docuRef.Notes        = 'Added from X++';
    docuRef.insert();

    info("Document note has been added successfully");
  }

}
```

3. Run the class to create the note.
4. Go back to the vendor list and click on the **Attachments** button in the form's **Action** pane or select **Document handling** from the **Command** menu under **File** to view the note added by our code, as shown in the following screenshot:



How it works...

All the document handling notes are stored in the `DocuRef` table, where three fields, `RefCompanyId`, `RefTableId`, and `RefRecId`, are used to identify the parent record. In this recipe, we set these fields to the vendor company ID, vendor table ID, and vendor account record ID, respectively. Then, we set the type, name, and description and inserted the document handling record. Notice that we have validated the document type before using it. In this way, we added a note to the record.

Using a normal table as a temporary table

Standard Dynamics 365 for Finance and Operations contains numerous temporary tables that are used by the application and can be used in custom modifications too. Although new temporary tables can also be easily created using the Dynamics 365 for Operations Project, sometimes it is not effective. One of the cases where it is not effective can be when the temporary table is similar to an existing one or exactly the same. The goal of this recipe is to demonstrate an approach for using standard non temporary tables in order to hold temporary data.

As an example, we will use the vendor table to insert and display a couple of temporary records without affecting the actual data.

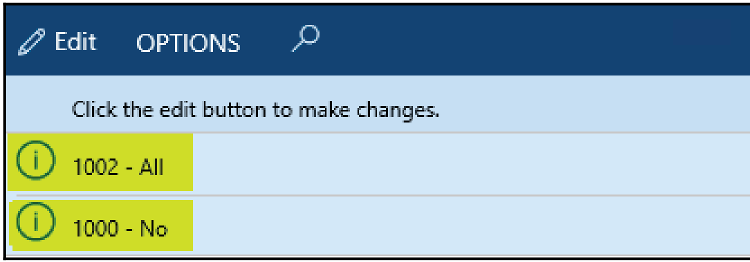
How to do it...

Carry out the following steps in order to complete this recipe:

1. In the Dynamics 365 Project, create a new class named `VendTableTmp` with the following code snippet:

```
 class VendTableTemp
{
  public static void main(Args _args)
 {
   VendTable   vendTable;

   vendTable.setTmp();

   vendTable.AccountNum = '1000';
   vendTable.Blocked    = CustVendorBlocked::No;
   vendTable.Party      = 1;
   vendTable.doInsert();
   vendTable.clear();
   vendTable.AccountNum = '1002';
   vendTable.Blocked    = CustVendorBlocked::All;
   vendTable.Party      = 2;
   vendTable.doInsert();

   while select vendTable
  {
    info(strFmt(
    "%1 - %2",
     vendTable.AccountNum,
      vendTable.Blocked));
  }
 }
}
```

2. Run the class and check the results, which may be similar to this:



How it works...

The key method in this recipe is `setTmp()`. This method is available in all the tables, and it makes the current table instance behave as a temporary table in the current scope. Basically, it creates an `InMemory` temporary table that has the same schema as the original table.

In this recipe, we create a new class and place all the code in its `main()` method. The reason why we create a class, not a job, is that the `main()` method can be set to run on the server tier by specifying the server modifier. This will improve the code's performance.

In the code, we first call the `setTmp()` method on the `vendTable` table to make it temporary in the scope of this method. This means that any data manipulations will be lost once the execution of this method is over and the actual table content will not be affected.

Next, we insert a couple of test records. Here, we use the `doInsert()` method to bypass any additional logic, which normally resides in the table's `insert()` method. We have to keep in mind that even the table becomes temporary; all the code in its `insert()`, `update()`, `delete()`, `initValue()`, and other methods is still present and we have to make sure that we don't call it unintentionally.

The last thing to do is to check for newly created records by listing the `vendTable` table. We can see that although the table contains many actual records, only the records that we inserted were displayed in the **Infolog** window. Additionally, the two records we inserted do not appear in the actual table.

Copying a record

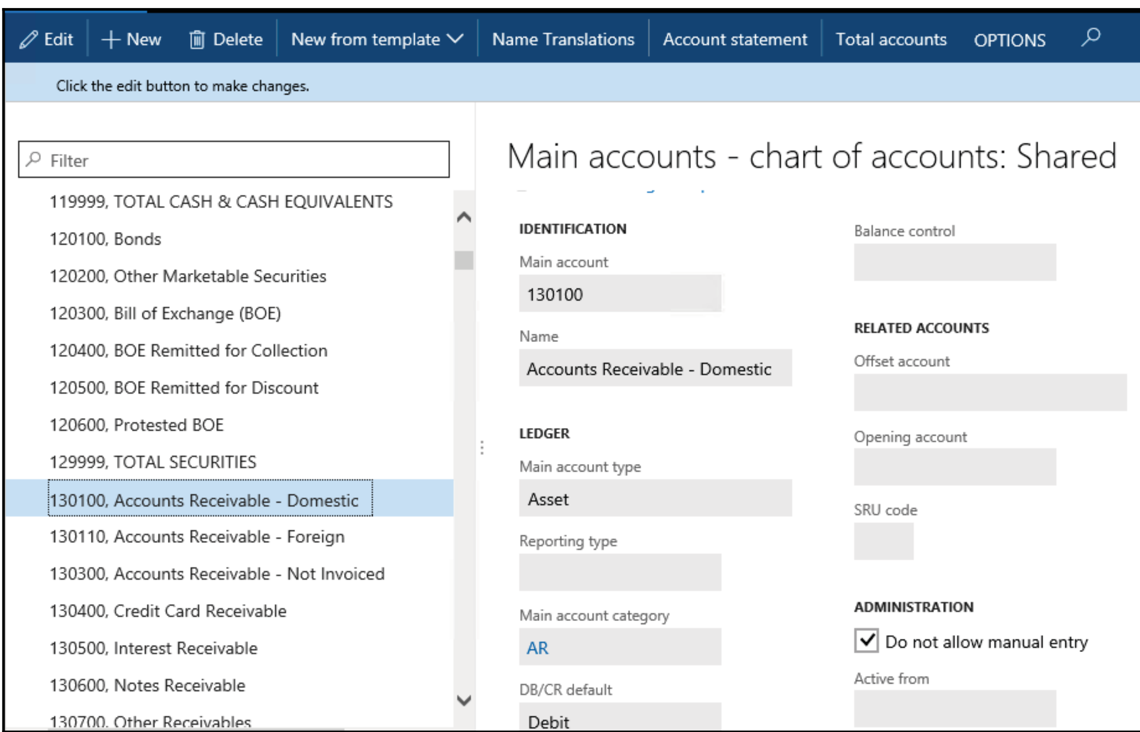
Copying existing data is one of the data manipulation tasks in Dynamics 365 for Finance and Operations. There are numerous places in the standard D365 application where users can create new data entries just by copying existing data and then modifying it. A few of the examples are the **Copy** button in **Cost management** | **Inventory accounting** | **Costing versions** and the **Copy project** button in **Project management** and **accounting** | **Projects** | **All projects**. Also, although the mentioned copying functionality might not be that straightforward, the idea is clear: the existing data is reused while creating new entries.

In this recipe, we will learn two ways to copy records in X++. We will discuss the usage of the table's `data()` method, the `global buf2buf()` function, and their differences. As an example, we will copy one of the existing ledger account records into a new record.

How to do it...

Carry out the following steps in order to complete this recipe:

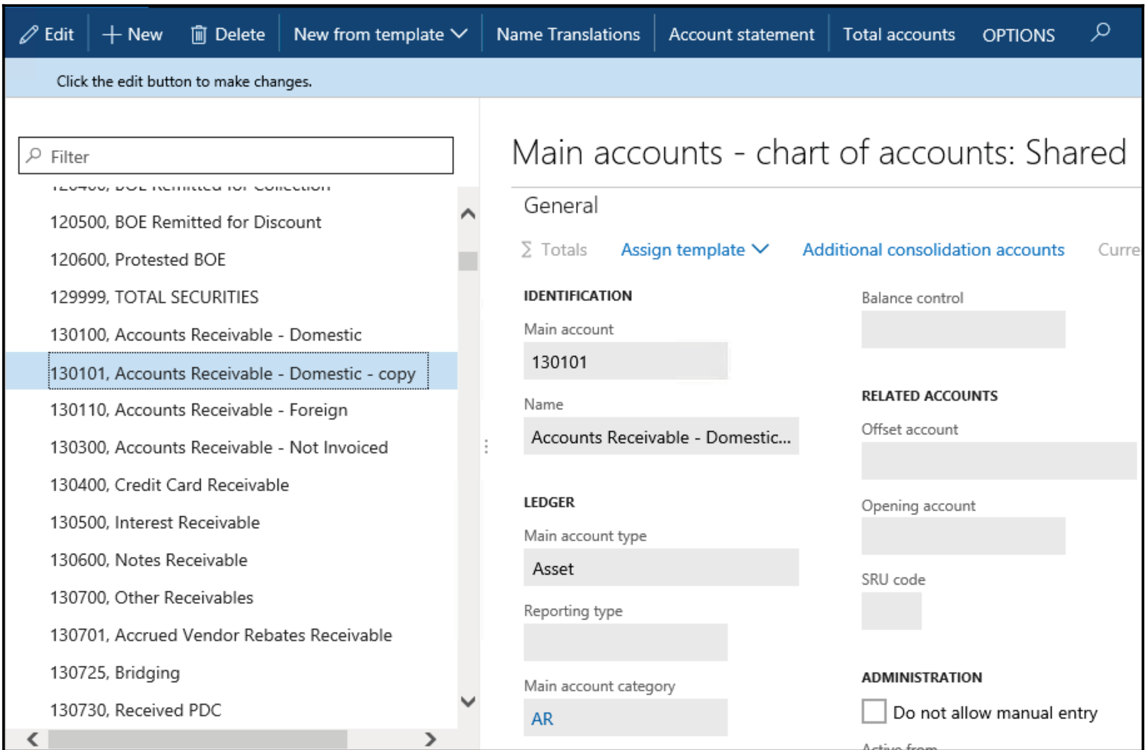
1. Navigate to **General ledger** | **Chart of accounts** | **Accounts** | **Main accounts** and find the account to be copied. In this example, we will use **130100**, as shown in the following screenshot:



2. Create a Dynamics 365 for Operations Project, create a runnable class named `MainAccountCopy` with the following code snippet, and run it:

```
class MainAccountCopy
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    MainAccount mainAccount1;
    MainAccount mainAccount2;

    mainAccount1 = MainAccount::findByMainAccountId(
    '130100');

    ttsBegin;
    mainAccount2.data(mainAccount1);
    mainAccount2.MainAccountId = '130101';
    mainAccount2.Name += ' - copy';

    if (!mainAccount2.validateWrite())
    {
      throw Exception::Error;
    }
    mainAccount2.insert();

    ttsCommit;
  }

}
```

3. Navigate to **General ledger** | **Chart of accounts** | **Accounts** | **Main accounts** again and notice that there are two identical records now, as shown in the following screenshot:



How it works...

In this recipe, we have two variables: `mainAccount1` for the original record and `mainAccount2` for the new record. First, we find the original record by calling `findMainAccountId()` in the `MainAccount` table.

Next, we copy it to the new one. Here, we use the `data()` table's `member` method, which copies all the data fields from one variable to another.

After that, we set a new ledger account number, which is a part of a unique table index.

Finally, we call `insert()` on the table if `validateWrite()` is successful. In this way, we create a new ledger account record, which is exactly the same as the existing one apart from the account number.

There's more...

As we saw before, the `data()` method copies all the table fields, including system fields such as the record ID, company account, and created user. Most of the time, it is OK because when the new record is saved, the system fields are overwritten with the new values. However, this function may not work for copying records across the companies. In this case, we can use another function called `buf2Buf()`. This function is a global function and is located in the `Global` class, which you can find by navigating to **AOT** | **Classes**. The `buf2Buf()` function is very similar to the table's `data()` method with one major difference. The `buf2Buf()` function copies all the data fields excluding the system fields. The code in the function is as follows:

```
static void buf2Buf(
 Common _from,
 Common _to,
 TableScope _scope = TableScope::CurrentTableOnly)
{
  DictTable   dictTable = new DictTable(_from.TableId);
  FieldId     fieldId   = dictTable.fieldNext(0, _scope);

  while (fieldId && ! isSysId(fieldId))
 {
    _to.(fieldId)   = _from.(fieldId);
    fieldId         = dictTable.fieldNext(fieldId, _scope);
 }
}
```

We can clearly see that during the copying process, all the table fields are traversed, but the system fields, such as `RecId` or `dataAreaId`, are excluded. The `isSysId()` helper function is used for this purpose.

In order to use the `buf2Buf()` function, the code of the `MainAccountCopy` job can be amended as follows:

```
 class MainAccountCopyBuf2Buf
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
   public static void main(Args _args)
  {
    MainAccount mainAccount1;
    MainAccount mainAccount2;

    mainAccount1 = MainAccount::findByMainAccountId('130100');

    ttsBegin;
    buf2Buf(mainAccount1, mainAccount2);

    mainAccount2.MainAccountId = '130102';
    mainAccount2.Name += ' - copy';

    if (!mainAccount2.validateWrite())
   {
     throw Exception::Error;
   }

    mainAccount2.insert();

    ttsCommit;
  }

}
```

Building a query object

Query objects in Dynamics 365 for Finance and Operations are used to build SQL statements for reports, views, forms, and so on. They are normally created in the AOT using the drag and drop functionality and by defining various properties. Query objects can also be created from the code at runtime. This is normally done when AOT tools cannot handle complex and/or dynamic queries.

In this recipe, we will create a query from the code to retrieve project records from the **Project management** module. We will select only the projects of the type **Time & material**, starting with **00005** in its number and containing at least one hour transaction. The project list will be sorted by project name.

How to do it...

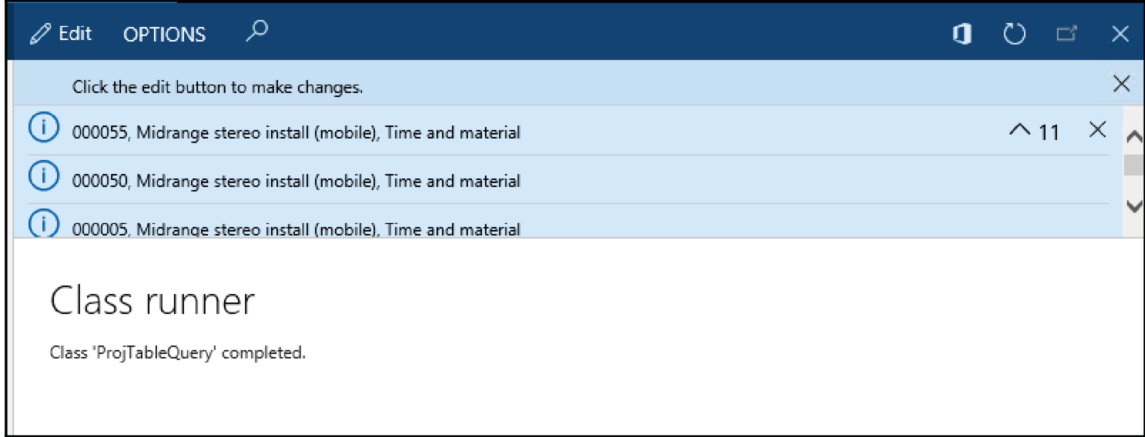
Carry out the following steps in order to complete this recipe:

1. Open the project area, create a runnable class named `ProjTableQuery`, and enter the following code snippet:

```
  class ProjTableQuery
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
 {
   Query                    query;
   QueryBuildDataSource     qbds1;
   QueryBuildDataSource     qbds2;
   QueryBuildRange          qbr1;
   QueryBuildRange          qbr2;
   QueryRun                 queryRun;
   ProjTable                projTable;

   query = new Query();

   qbds1 = query.addDataSource(tableNum(ProjTable));
   qbds1.addSortField(
   fieldNum(ProjTable, Name),
   SortOrder::Ascending);

   qbr1 = qbds1.addRange(fieldNum(ProjTable,Type));
   qbr1.value(queryValue(ProjType::TimeMaterial));

   qbr2 = qbds1.addRange(fieldNum(ProjTable,ProjId));
   qbr2.value(
    SysQuery::valueLike(queryValue('00005')));

   qbds2 = qbds1.addDataSource(tableNum(ProjEmplTrans));
   qbds2.relations(true);
   qbds2.joinMode(JoinMode::ExistsJoin);
```

```
            queryRun = new QueryRun(query);

            while (queryRun.next())
           {
             projTable = queryRun.get(tableNum(ProjTable));
              info(strFmt(
              "%1, %2, %3",
              projTable.ProjId,
              projTable.Name,
              projTable.Type));
           }
         }
       }
```

2. Run the class and you will get a screen similar to the following screenshot:



How it works...

First, we create a new `query` object. **Next**, we add a new `ProjTable` data source to the `query` object by calling its `addDataSource()` member method. The method returns a reference to the `QueryBuildDataSource` object-`qbds1`. Here, we call the `addSortField()` method to enable sorting by the project name.

The following two blocks of code create two ranges. The first block of code shows only the projects of the `time & material` type and the second one lists only the records where the project number starts with `00005`. These two filters are automatically added together using SQL's `AND` operator. The `QueryBuildRange` objects are created by calling the `addRange()` member method of the `QueryBuildDataSource` object with the field ID number as the argument. The range value is set by calling `value()` on the `QueryBuildRange` object itself. We use the `queryValue()` function from the `Global` class and the `valueLike()` function from the `SysQuery` class to prepare the values before applying them as a range. More functions, such as `queryNotValue()` and `queryRange()`, can be found in the `Global` application class by navigating to **AOT** | **Classes**. Note that these functions are actually shortcuts to the `SysQuery` application class, which in turn has even more interesting helper methods that might be handy for every developer.

Adding another data source to an existing one connects both the data sources using SQL's `JOIN` operator. In this example, we are displaying projects that have at least one posted hour line. We start by adding the `ProjEmplTrans` table as another data source.

Next, we need to add relationships between the tables. If relationships are not defined on tables, we will have to use the `addLink()` method with relation field's ID numbers. In this example, relations in the tables are already defined, so you only need to enable them by calling the `relations()` method with `true` as an argument.

Calling `joinMode()` with `JoinMode::ExistsJoin` as a parameter ensures that only the projects that have at least one hour transaction will be selected. In situations like this, where we do not need any data from the second data source, performance-wise it is better to use an `exists` join instead of the `inner` join. This is because the `inner` join fetches the data from the second data source and, therefore, takes longer to execute.

The last thing that needs to be done is to create and run the `queryRun` object and show the selected data on the screen.

There's more...

It is worth mentioning a couple of specific cases when working with query objects from the code. One of them is how to use the `OR` operator and the other one is how to address array fields.

Using the OR operator

As you have already noted, regardless of how many ranges are added, all of them will be added together using SQL's `AND` operator. In most cases, this is fine, but sometimes complex user requirements demand ranges to be added using SQL's `OR` operator. There might be a number of workarounds, such as using temporary tables or similar tools, but we can use the Dynamics 365 for Operations feature that allows you to pass a part of a raw SQL string as a range.

In this case, the range has to be formatted in a manner similar to a fully-qualified SQL `where` clause, including field names, operators, and values. The expressions have to be formatted properly before you use them in a query. Here are some of the rules:

- The expression must be enclosed within single quotes
- Inside, the whole expression has to be enclosed within parentheses
- Each subexpression must also be enclosed within parentheses
- String values have to be enclosed within double quotes
- For enumerations, use their numeric values

For value formatting, use various Dynamics 365 for Operations functions, such as `queryValue()` and `date2StrXpp()`, or methods from the `SysQuery` class.

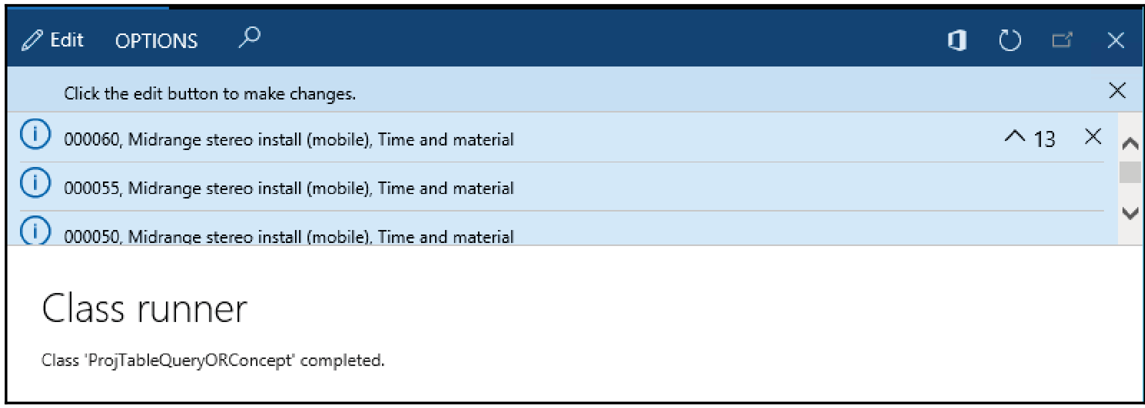
Let's replace the code snippet from the previous example with the following lines of code:

```
qbr2.value(SysQuery::valueLike (queryValue('00005')));
with the new code:
qbr2.value(strFmt('((%1 like "%2") || (%3 = %4))',
 fieldStr(ProjTable,ProjId),queryvalue('00005*'),
  fieldStr(ProjTable,Status),ProjStatus::InProcess+0));
```

Notice that by adding zero to the enumeration in the previous code, we can force the `strFmt()` function to use the numeric value of the enumeration. The `strFmt()` output should be similar to the following line:

```
((ProjId like "00005*") || (Status = 3))
```

Now if you run the code, besides all the projects starting with `00005`, the result will also include all the active projects, as shown in the following screenshot:



See also

- The *Creating a custom filter* recipe in `Chapter 3`, *Working with Data in Forms*
- The *Using a form for building a lookup* recipe in `Chapter 4`, *Building Lookups*

Using a macro in a SQL statement

In a standard Dynamics 365 for Finance and Operations application, there are macros, such as `InventDimJoin` and `InventDimSelect`, which are reused numerous times across the application. These macros are actually full or partial X++ SQL queries that can be called with various arguments. Such approaches save development time by allowing you to reuse pieces of X++ SQL queries.

In this recipe, we will create a small macro, which holds a single `where` clause, to display only the active vendor records. Then, we will create a class that uses the created macros to display a vendor list.

How to do it...

Carry out the following steps in order to complete this recipe:

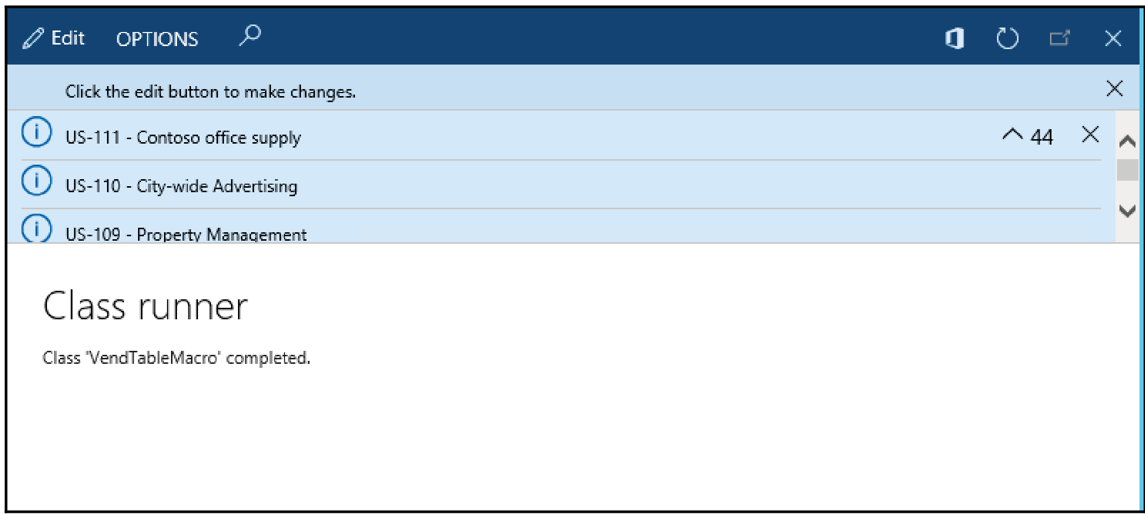
1. Create a Dynamics 365 for Operations Project and create a new macro named `VendTableNotBlocked` with the following code snippet:

```
(%1.Blocked == CustVendorBlocked::No)
```

2. In the Dynamics 365 Project, create a new `runnable` class called `VendTableMacro` with the following code:

```
class VendTableMacro
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
 {
   VendTable   vendTable;

   while select vendTable
   where #VendTableNotBlocked(vendTable)
  {
     info(strFmt(
     "%1 - %2",
     vendTable.AccountNum,
     vendTable.name()));
  }
 }

}
```

3. Run the job and check the results, as shown in the following screenshot:



How it works...

First, we define a macro that holds the `where` clause. Normally, the purpose of defining SQL in a macro is to reuse it a number of times in various places. We use `%1` as an argument. More arguments can be used.

Next, we create a job with the `select` statement. Here, we use the previously created macro in the `where` clause and pass `vendTable` as an argument.

The query works like any other query, but the advantage is that the code in the macro can be reused elsewhere.

Remember that before we start using macros in SQL queries, we should be aware of the following caveats:

- Too much code in a macro might reduce the SQL statement's readability for other developers
- Cross-references do not take into account the code inside the macro
- Changes in the macro will not reflect in the objects where the macro is used until the objects are recompiled

Executing a direct SQL statement

Dynamics 365 for Finance and Operations allows developers to build X++ SQL statements that are flexible enough to fit into any custom business process. However, in some cases, the usage of X++ SQL is either not effective or not possible at all. One such case is when we run data upgrade tasks during an application version upgrade. A standard application contains a set of data upgrade tasks to be completed during the version upgrade. If the application is highly customized, then most likely, standard tasks have to be modified in order to reflect data dictionary customization's, or a new set of tasks have to be created to make sure data is handled correctly during the upgrade.

Normally, at this stage, SQL statements are so complex that they can only be created using database-specific SQL and executed directly in the database. Additionally, running direct SQL statements dramatically increases data upgrade performance because most of the code is executed on the database server where all the data resides. This is very important while working with large volumes of data.

This recipe will demonstrate how to execute SQL statements directly. We will connect to the current Dynamics 365 for Finance and Operations database directly using an additional connection and retrieve a list of vendor accounts.

How to do it...

Carry out the following steps in order to complete this recipe:

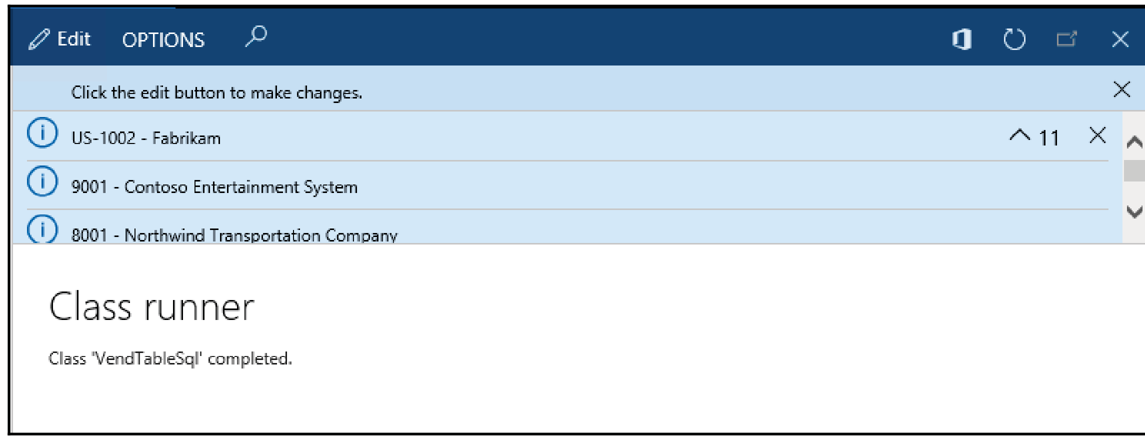
1. In the Dynamics 365 Project, create a new class named `VendTableSql` using the following code snippet:

```
class VendTableSql
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    UserConnection                  userConnection;
    Statement                       statement;
    str                             sqlStatement;
    SqlSystem                       sqlSystem;
    SqlStatementExecutePermission   sqlPermission;
    ResultSet                       resultSet;
    DictTable                       tblVendTable;
```

```
DictTable                        tblDirPartyTable;
DictField                        fldParty;
DictField                        fldAccountNum;
DictField                        fldDataAreaId;
DictField                        fldBlocked;
DictField                        fldRecId;
DictField                        fldName;
tblVendTable     = new DictTable(tableNum(VendTable));
tblDirPartyTable = new DictTable(tableNum(DirPartyTable));

fldParty = new DictField(
 tableNum(VendTable),
 fieldNum(VendTable,Party));

fldAccountNum = new DictField(
 tableNum(VendTable),
 fieldNum(VendTable,AccountNum));

fldDataAreaId = new DictField(
 tableNum(VendTable),
 fieldNum(VendTable,DataAreaId));

fldBlocked = new DictField(
 tableNum(VendTable),
 fieldNum(VendTable,Blocked));

fldRecId = new DictField(
 tableNum(DirPartyTable),
 fieldNum(DirPartyTable,RecId));

fldName = new DictField(
 tableNum(DirPartyTable),
 fieldNum(DirPartyTable,Name));

sqlSystem = new SqlSystem();

sqlStatement = 'SELECT %1, %2 FROM %3 ' +
'JOIN %4 ON %3.%5 = %4.%6 ' +
'WHERE %7 = %9 AND %8 = %10';

sqlStatement = strFmt(
 sqlStatement,
 fldAccountNum.name(DbBackend::Sql),
 fldName.name(DbBackend::Sql),
 tblVendTable.name(DbBackend::Sql),
 tblDirPartyTable.name(DbBackend::Sql),
 fldParty.name(DbBackend::Sql),
 fldRecId.name(DbBackend::Sql),
```

```
            fldDataAreaId.name(DbBackend::Sql),
            fldBlocked.name(DbBackend::Sql),
            sqlSystem.sqlLiteral(curext(), true),
            sqlSystem.sqlLiteral(CustVendorBlocked::No, true));

           userConnection = new UserConnection();
           statement      = userConnection.createStatement();

           sqlPermission = new SqlStatementExecutePermission(
            sqlStatement);

           sqlPermission.assert();

           resultSet      = statement.executeQuery(sqlStatement);

           CodeAccessPermission::revertAssert();

           while (resultSet.next())
          {
            info(strFmt(
            "%1 - %2",
            resultSet.getString(1),
            resultSet.getString(2)));
          }
        }
      }
```

2. Run the class to retrieve a list of vendors directly from the database, as shown in the following screenshot:



How it works...

We start the code by creating the `DictTable` and `DictField` objects to handle the vendor table and its fields, which are used later in the query. The `DirPartyTable` is used to get additional vendor information.

A new `SqlSystem` object is also created. It is used to convert D365 types to SQL types.

Next, we set up a SQL statement with a number of placeholders for the table or field names and field values to be inserted later.

The main query creation takes place next, when the query placeholders are replaced with the right values. Here, we use the previously created `DictTable` and `DictField` type objects by calling their `name()` methods with the `DbBackend::Sql` enumeration as an argument. This ensures that we pass the name in the exact manner it is used in the database-some of the SQL field names are not necessary, which is the same as field names within the application.

We also use the `sqlLiteral()` method of the previously created `sqlSystem` object to properly format SQL values in order to ensure that they do not have any unsafe characters.

The value of the `sqlStatement` variable that holds the prepared SQL query depending on your environment is as follows:

```
SELECT ACCOUNTNUM, NAME FROM VENDTABLE
JOIN DIRPARTYTABLE ON VENDTABLE.PARTY = DIRPARTYTABLE.RECID
WHERE DATAAREAID = 'usmf' AND BLOCKED = 0
```

Once the SQL statement is ready, we initialize a direct connection to the database and run the statement. The results are returned in the `resultSet` object, and we get them by using the `while` statement and calling the `next()` method until the end.

Note that we created an `sqlPermission` object of the type `SqlStatementExecutePermission` here and called its `assert()` method before executing the statement. This is required in order to comply with Dynamics 365 for Operation's trustworthy computing requirements.

Another thing that needs to be mentioned is that when building direct SQL queries, special attention has to be paid to license, configuration, and security keys. Some tables or fields might be disabled in the application and may contain no data in the database.

The code in this recipe can be also used to connect to external ODBC databases. We only need to replace the `UserConnection` class with the `OdbcConnection` class and use text names instead of the `DictTable` and `DictField` objects.

There's more...

The standard Dynamics 365 for Finance and Operations application provides an alternate way of building direct SQL statements by using a set of `SQLBuilder` classes. By using these classes, we can create SQL statements as objects, as opposed to text. Next, we will demonstrate how to use a set of `SQLBuilder` classes. We will create the same SQL statement as we did before.

First, in a Dynamics 365 project, create another class named `VendTableSqlBuilder` using the following code snippet:

```
 class VendTableSqlBuilder
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
   public static void main(Args _args)
  {
    UserConnection                  userConnection;
    Statement                       statement;
    str                             sqlStatement;
    SqlStatementExecutePermission   sqlPermission;
    ResultSet                       resultSet;
    SQLBuilderSelectExpression      selectExpr;
    SQLBuilderTableEntry            vendTable;
    SQLBuilderTableEntry            dirPartyTable;
    SQLBuilderFieldEntry            accountNum;
    SQLBuilderFieldEntry            dataAreaId;
    SQLBuilderFieldEntry            blocked;
    SQLBuilderFieldEntry            name;

    selectExpr = SQLBuilderSelectExpression::construct();
    selectExpr.parmUseJoin(true);

    vendTable = selectExpr.addTableId(
     tablenum(VendTable));

    dirPartyTable = vendTable.addJoinTableId(
     tablenum(DirPartyTable));

    accountNum = vendTable.addFieldId(
     fieldnum(VendTable,AccountNum));

    name = dirPartyTable.addFieldId(
     fieldnum(DirPartyTable,Name));
```

```
        dataAreaId = vendTable.addFieldId(
         fieldnum(VendTable,DataAreaId));

        blocked = vendTable.addFieldId(
         fieldnum(VendTable,Blocked));
        vendTable.addRange(dataAreaId, curext());
         vendTable.addRange(blocked, CustVendorBlocked::No);

        selectExpr.addSelectFieldEntry(
         SQLBuilderSelectFieldEntry::newExpression(
          accountNum,
          'AccountNum'));

        selectExpr.addSelectFieldEntry(
         SQLBuilderSelectFieldEntry::newExpression(
          name, 'Name'));

        sqlStatement   = selectExpr.getExpression(null);

        userConnection = new UserConnection();
        statement      = userConnection.createStatement();

        sqlPermission = new SqlStatementExecutePermission(
         sqlStatement);

        sqlPermission.assert();

        resultSet = statement.executeQuery(sqlStatement);

        CodeAccessPermission::revertAssert();

        while (resultSet.next())
       {
         info(strfmt(
         "%1 - %2",
          resultSet.getString(1),
          resultSet.getString(2)));
       }
      }

    }
```

In the preceding method, we first create a new `selectExpr` object, which is based on the `SQLBuilderSelectExpression` class. It represents the object of the SQL statement.

Next, we add the `VendTable` table to it by calling its member method `addTableId()`. This method returns a reference to the `vendTable` object of the type `SQLBuilderTableEntry`, which corresponds to a table node in a SQL query. We also add `DirPartyTable` as a joined table.

Then, we create a number of field objects of the `SQLBuilderFieldEntry` type to be used later and two ranges to show only this company account and only the active vendor accounts.

We use `addSelectFieldEntry()` to add two fields to be selected. Here, we use the previously created field objects.

The SQL statement is generated once the `getExpression()` method is called, and the rest of the code is the same as in the previous example.

Running the class will give us results, which are exactly similar to the ones we got earlier.

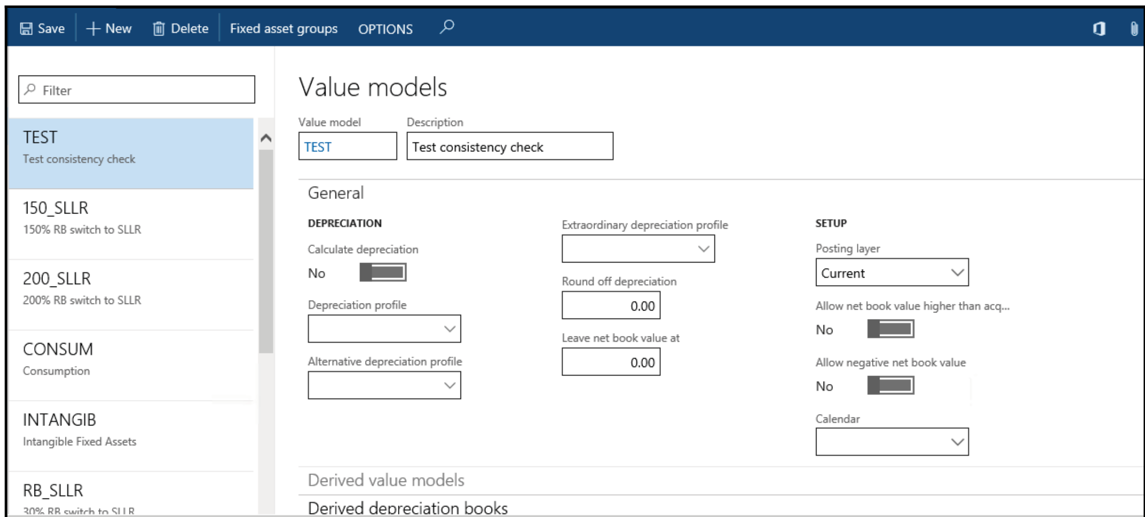
Enhancing the data consistency checks

It is highly recommended that you run the standard Dynamics 365 for Finance and Operations data consistency checks from time to time, which can be found by navigating to **System administration** | **Periodic tasks** | **Database** | **Consistency check**, to check the system's data integrity. This function finds orphan data, validates parameters, and does many other things, but it does not do everything. The good thing is that it can be easily extended.

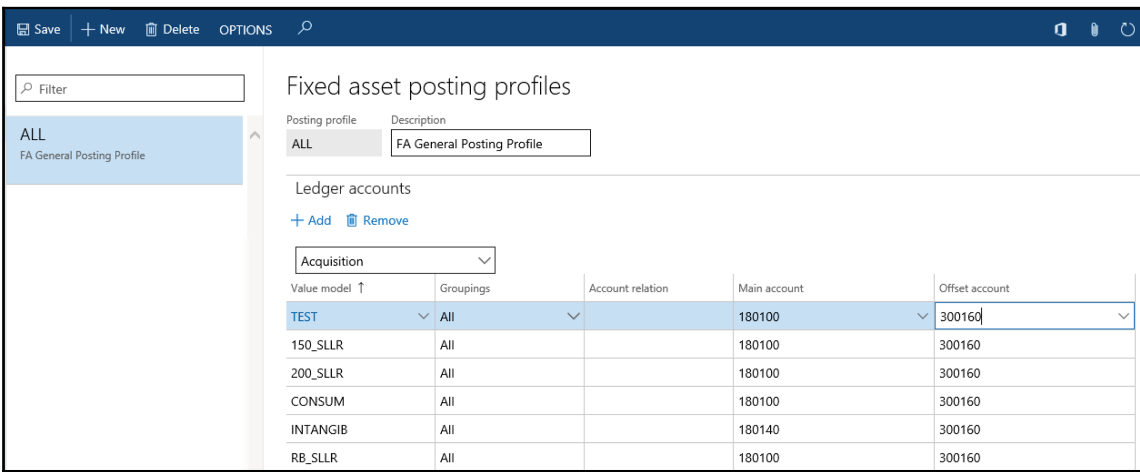
In this recipe, we will see how we can enhance the standard Dynamics 365 for Finance and Operations consistency check to include more tables in its data integrity validation.

Getting ready

Before we start, we need to create an invalid setup in order to make sure that we can simulate data inconsistency. Navigate to **Fixed assets** | **Setup** | **Value models** and create a new model, for instance, `TEST`, as shown in the following screenshot:



Navigate to **Fixed assets** | **Setup** | **Fixed asset posting profiles** and under the **Ledger accounts** group, create a new record with the newly created value model for any of the posting types, as shown here:



Go back to the **Value models** form and delete the previously created value model. Now, we have a nonexistent value model in the fixed asset posting settings.

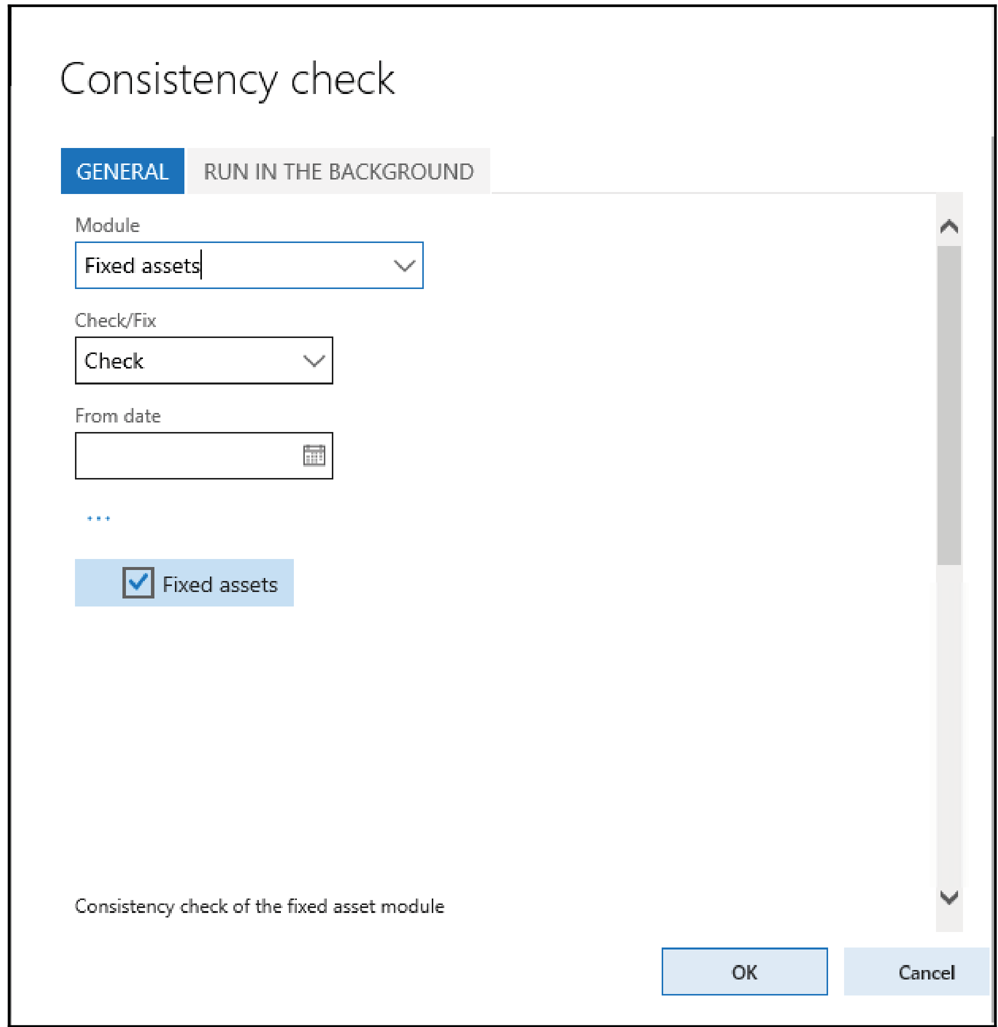
How to do it...

Carry out the following steps in order to complete this recipe:

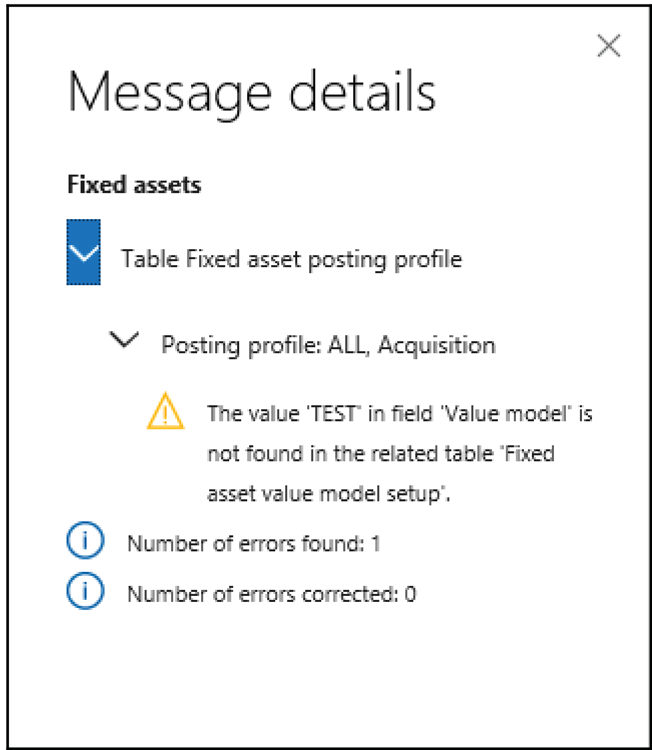
1. In the Dynamics 365 Project, create a new class named `AssetConsistencyCheck` with the following code snippet:

```
class AssetConsistencyCheck extends SysConsistencyCheck
{
   client server static ClassDescription description()
    {
      return "Fixed assets";
    }

    client server static HelpTxt helpText()
    {
      return "Consistency check of the fixed asset module";
    }

    public Integer executionOrder()
    {
      return 1;
    }

    public void run()
    {
      this.kernelCheckTable(tableNum(AssetLedgerAccounts));
    }

}
```

2. Navigate to **System administration** | **Periodic tasks** | **Database** | **Consistency check**, select the newly created **Fixed assets** option from the **Module** drop-down list, and click on **OK** to run the check, as shown here:



3. Now, the message displayed in the **Infolog** window should complain about the missing value model in the fixed assets posting settings, as shown in the following screenshot:



How it works...

The consistency check in Dynamics 365 for Finance and Operations validates only the predefined list of tables for each module. The system contains a number of classes derived from `SysConsistencyCheck`. For example, the `CustConsistencyCheck` class is responsible for validating the **Accounts receivable** module, `LedgerConsistencyCheck` for validating **General ledger**, and so on.

In this recipe, we created a new class named `AssetConsistencyCheck`, extending the `SysConsistencyCheck` class for the fixed asset module. The following methods were created:

- `description()`: This provides a name to the consistency check form.
- `helpText()`: This displays some explanation about the check.
- `executionOrder()`: This determines where the check is located in the list.
- `run()`: This holds the code to perform the actual checking. Here, we use the `kernelCheckTable()` member method, which validates the given table.

There's more...

The classes that we just mentioned can only be executed from the main **Consistency check** form. Individual checks can also be invoked as standalone functions. We just need to create an additional method to allow the running of the class:

```
 static void main(Args _args)
{
  SysConsistencyCheckJob consistencyCheckJob;
  AssetConsistencyCheck  assetConsistencyCheck;

  consistencyCheckJob = new SysConsistencyCheckJob(
   classIdGet(assetConsistencyCheck));

  if (!consistencyCheckJob.prompt())
 {
   return;
 }

  consistencyCheckJob.run();
}
```

Using the date effectiveness feature

Date effectiveness allows developers to easily create date range fields. Date ranges are used to define record validity between the specified dates, for example, defining employee contract dates and defining vendor license validity.

This feature significantly reduces the amount of time that developers spend on developing business logic/code and also provides a consistent approach to implement data range fields.

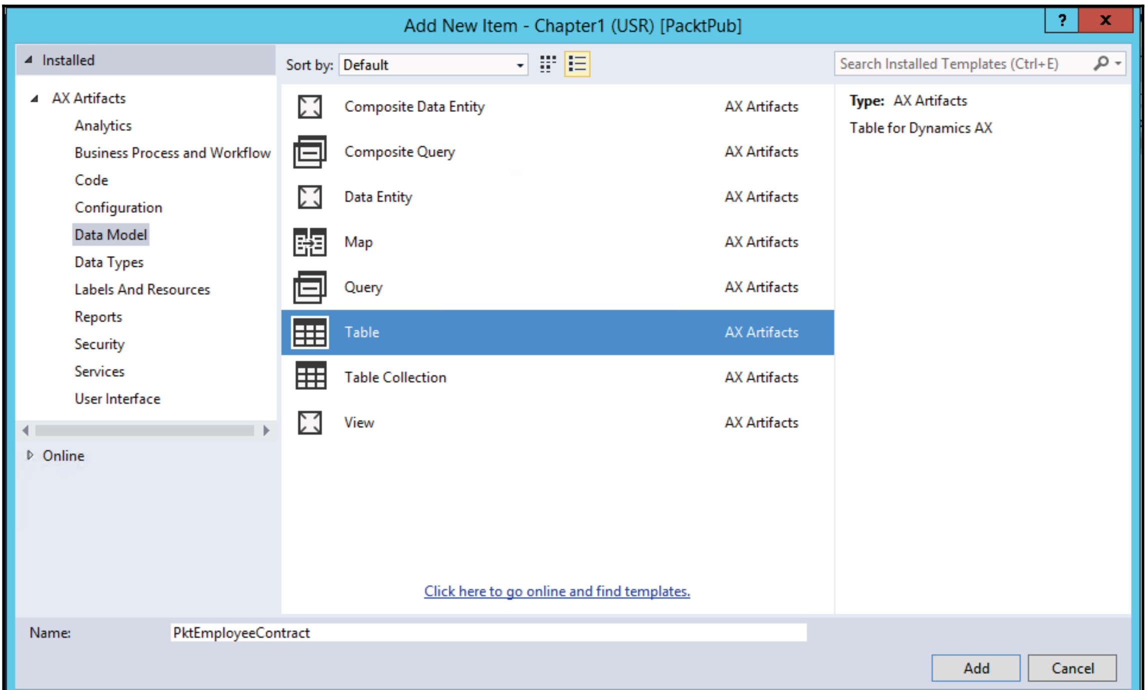
This recipe will demonstrate the basics of date effectiveness. We will create a new table to implement date range validation.

How to do it...

Carry out the following steps in order to complete this recipe:

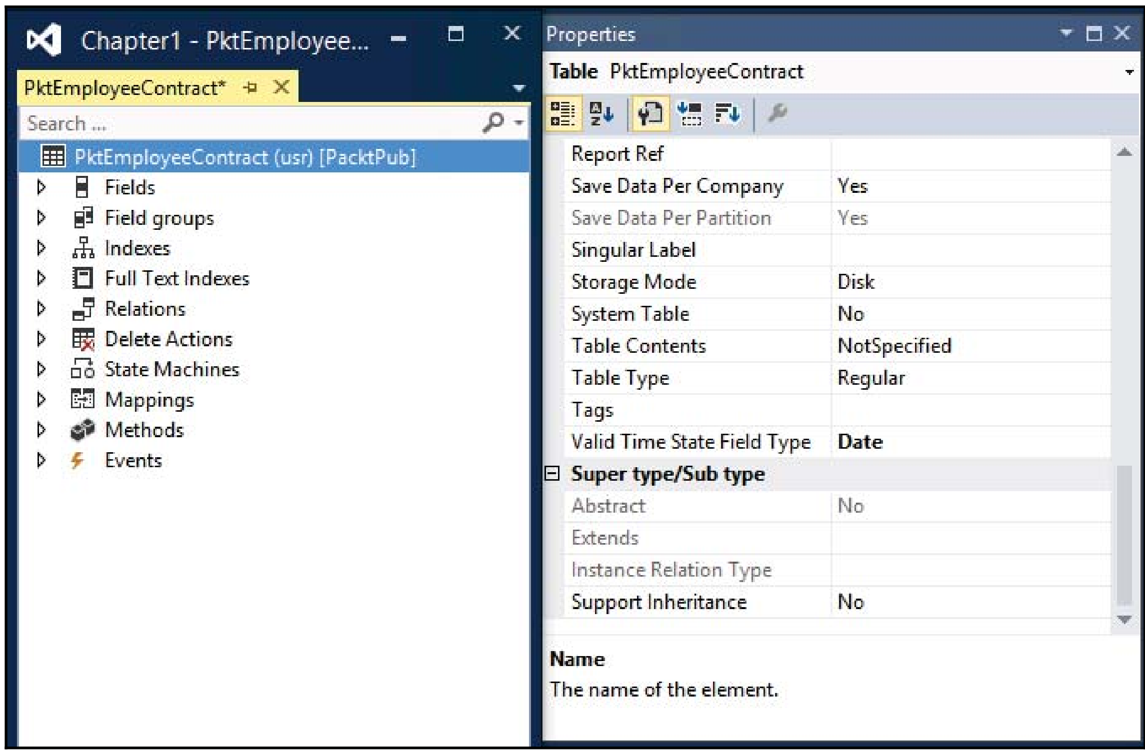
Run Visual Studio as admin:

1. Load your earlier project.
2. Add a new **TablePktEmployeeContract.**

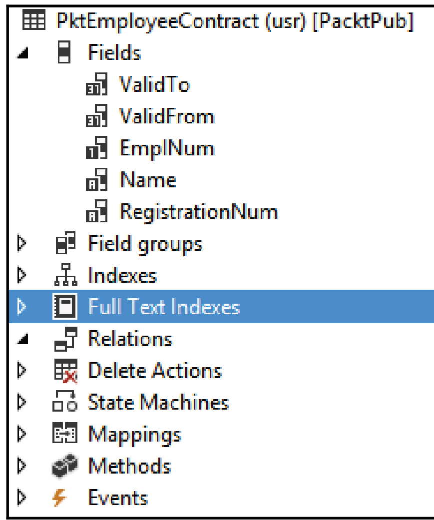


Set the property as follows:

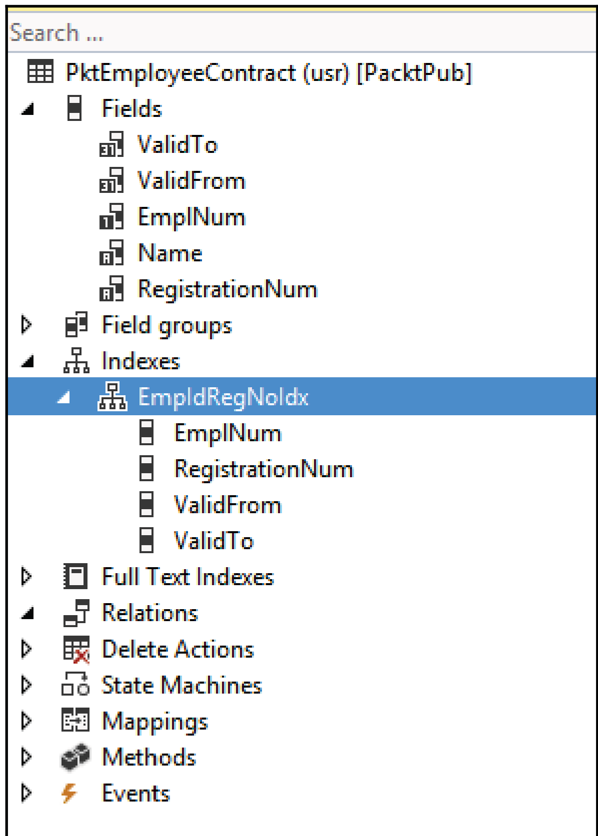
Property	Value
ValidTimeStateFieldType	Date



Note the two new fields that are automatically added to the table, as shown in the following screenshot (**ValidTo** and **ValidFrom**):



3. Now create a new index as follows and add fields as follows:



4. Set the following mentioned property for the index here:

Property	**Value**
AlternateKey	Yes
ValidTimeStateKey	Yes
ValidTimeStateMode	NoGap

5. Now open the table and enter some records in this table itself instead of creating a new form for the table. Right-click on **Table** and select **Browse table**:

How it works...

We start the recipe by setting the `ValidTimeStateFieldType` property to `Date` in the `SysEmailTable` table. This automatically creates two new fields--`ValidFrom` and `ValidTo` that are used to define a date range.

Next, we add the created fields to the primary index where the `EmplNum` field is used and adjust the index's properties.

We set the `AlternateKey` property to `Yes` in order to ensure that this index is a part of an alternate key.

We set the `ValidTimeStateKey` property to `Yes` in order to specify that the index is used to determine valid date ranges.

We also set the `ValidTimeStateMode` property to `NoGap` in order to ensure that email templates with the same identification number can be created within continuous periods. This property can also be set to `Gap`, allowing noncontiguous date ranges.

2

Working with Forms

In this chapter, we will cover the following recipes:

- Creating dialogs using the RunBase framework
- Handling the dialog event
- Creating dialogs using the SysOperation framework
- Building a dynamic form
- Adding a form splitter
- Creating a modal form
- Modifying multiple forms dynamically
- Storing the last form values
- Using a Tree control
- Adding the View details link
- Selecting a Form Pattern
- Full list of form patterns
- Creating a new form

Introduction

Forms in Dynamics 365 for Finance and Operations represent the user interface and are mainly used to enter or modify data. They are also used to run reports, execute user commands, validate data, and so on.

Normally, forms are created using the AOT by producing a form object and adding form controls, such as tabs, tab pages, grids, groups, data fields, and images. The form's behavior is controlled by its properties or the code in its member methods. The behavior and layout of form controls are also controlled by their properties and the code in their member methods. Although it is very rare, forms can also be created dynamically from code.

In this chapter, we will cover various aspects of using Dynamics 365 for Finance and Operations forms. We start by building Dynamics 365 for Finance and Operations dialogs, which are actually dynamic forms, and then go on to explain how to handle their events. The chapter will also show you how to build dynamic forms, how to add dynamic controls to existing forms, and how to make modal forms.

Creating dialogs using the RunBase framework

Dialogs are a way to present users with a simple input form. They are commonly used for small user tasks, such as filling in report values, running batch jobs, and presenting only the most important fields to the user when creating a new record. Dialogs are normally created from X++ code without storing the actual layout in the AOT.

The application class called `Dialog` is used to build dialogs. Other application classes, such as `DialogField`, `DialogGroup`, and `DialogTabPage`, are used to create dialog controls. The easiest way to create dialogs is to use the `RunBase` framework. This is because the framework provides a set of predefined methods, which make the creation and handling of the dialog well-structured, as opposed to having all the code in a single place.

In this example, we will demonstrate how to build a dialog from code using the `RunBase` framework class. The dialog will contain customer table fields shown in different groups and tabs for creating a new record. There will be two tab pages, `General` and `Details`. The first page will have the `Customer account` and `Name` input controls. The second page will be divided into two groups, `Setup` and `Payment`, with relevant fields inside each group. The actual record will not be created, as it is beyond the scope of this example. However, for demonstration purposes, the information specified by the user will be displayed in the **Infolog** window.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Add a new project `Create dialog`.
2. Add a new `Runnable` class and rename it `MyDialog`. Now, add the following code snippet:

   Declare all your objects in the class, as shown follows:

```
class MyDialog extends RunBase
{
   DialogField    fieldAccount;
   DialogField    fieldName;
   DialogField    fieldGroup;
   DialogField    fieldCurrency;
   DialogField    fieldPaymTermId;
   DialogField    fieldPaymMode;
   CustName       custName;
   CustGroupId    custGroupId;
   CurrencyCode   currencyCode;
   CustPaymTermId paymTermId;
   CustPaymMode   paymMode;

 public container pack()
    {
       return conNull();
    }

    public boolean unpack(container _packedClass)
    {
      return true;
    }
```

- Create a dialog method to capture runtime user inputs for customer details:

```
    Object dialog()
    {
       Dialog           dialog;
       DialogGroup      groupCustomer;
       DialogGroup      groupPayment;

       dialog = super();

       dialog.caption("Customer information");

       fieldAccount=dialog.addField
```

```
    (extendedTypeStr(CustVendAC),"Customer account");

    fieldName =dialog.addField(extendedTypeStr(CustName));

    dialog.addTabPage("Details");

    groupCustomer  = dialog.addGroup("Setup");
    fieldGroup=dialog.addField
     (extendedTypeStr(CustGroupId));
    fieldCurrency=dialog.addField
     (extendedTypeStr(CurrencyCode));

    groupPayment    = dialog.addGroup("Payment");
    fieldPaymTermId=dialog.addField
     (extendedTypeStr(CustPaymTermId));
    fieldPaymMode = dialog.addField
     (extendedTypeStr(CustPaymMode));

    return dialog;
}
```

- Now, when users select their desired values, we need to read all of them to show in the infolog. Use `getFromDialog` to read a dialog field's value:

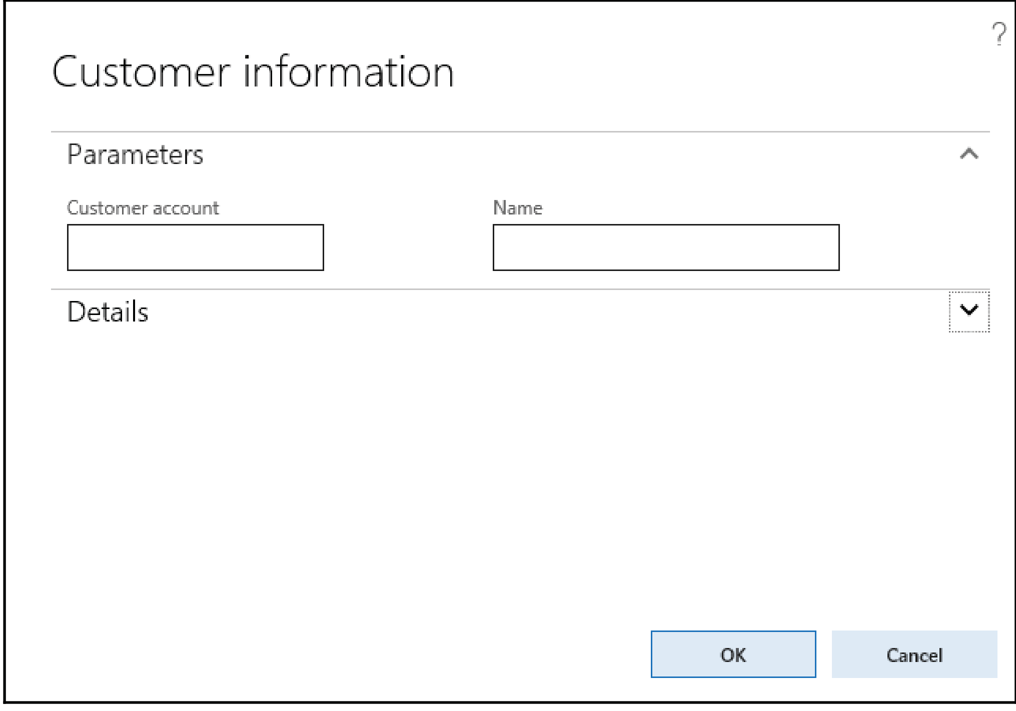
```
public boolean getFromDialog()
{
    custAccount  = fieldAccount.value();
    custName     = fieldName.value();
    custGroupId  = fieldGroup.value();
    currencyCode = fieldCurrency.value();
    paymTermId   = fieldPaymTermId.value();
    paymMode     = fieldPaymMode.value();
    return super();

}
```

- Use the `run` method to make Infolog statements, as in the following code:

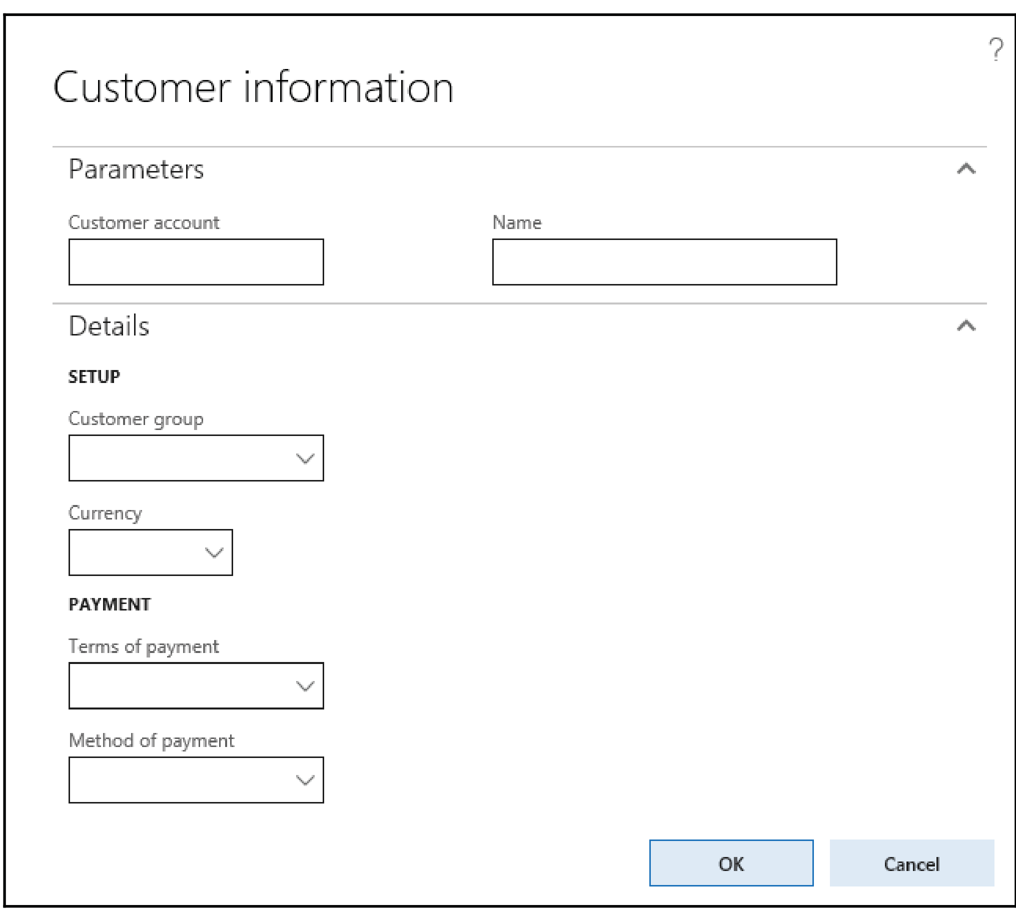
```
public void run()
{
    info("You have entered customer information:");
    info(strFmt("Account: %1", custAccount));
    info(strFmt("Name: %1", custName));
    info(strFmt("Group: %1", custGroupId));
    info(strFmt("Currency: %1", currencyCode));
    info(strFmt("Terms of payment: %1", paymTermId));
    info(strFmt("Method of payment: %1", paymMode));
}
```

```
        public static void main(Args _args)
        {
            MyDialog    myDialog = new MyDialog();

             if (myDialog.prompt())
             {
                myDialog.run();
             }
        }
    }
```

3. In order to test the dialog, right-click on this class and **set as startup project**.
4. Build your project. Now, run the project. The following form will appear in the internet browser:



4. Click on the **Details** tab page; you will see a screen similar to the following screenshot:



5. Enter information in all the fields and click on **OK**. The results will be displayed on the **Infolog** tab on top of the browser window.

How it works...

First, we create a new class named `MyDialog`. By extending it from `RunBase`, we utilize a standard approach to develop data manipulation functions in Dynamics 365 for Operations . The `RunBase` framework will define a common structure and automatically add additional controls, such as the **OK** and **Cancel** buttons, to the dialog.

Then, we declare class member variables, which will be used later. The `DialogField` type variables are actual user input fields. The rest of the variables are used to store the values returned from the user input.

The `pack()` and `unpack()` methods are normally used to convert an object into a container and convert the container back into an object, respectively. A container is a common format used to store objects in the user cache (`SysLastValue`) or to transfer the object between the server and client tiers. The `RunBase` framework needs these two methods to be implemented in all its subclasses. In this example, we are not using any of the `pack()` or `unpack()` features, but because these methods are mandatory, we return an empty container from `pack()` and we return `true` from `unpack()`.

The layout of the actual dialog is constructed in the `dialog()` member method. Here, we define local variables for the dialog itself-tab pages and groups. These variables, as opposed to the dialog fields, do not store any values for further processing. The `super()` in the method creates the initial dialog object for us and automatically adds the relevant controls, including the **OK** and **Cancel** buttons.

Additional dialog controls are added to the dialog by using the `addField()`, `addGroup()`, and `addTabPage()` methods. There are more methods, such as `addText()`, `addImage()`, and `addMenuItemButton()`, which are used to add different types of controls. All the controls have to be added to the dialog object directly. Adding an input control to groups or tabs is done by calling `addField()` right after `addGroup()` or `addTabPage()`. In the previous example, we added tab pages, groups, and fields in a top-down logical sequence. Note that it is enough only to add a second tab page; the first tab page, labeled `General`, is added automatically by the `RunBase` framework.

Values from the dialog controls are assigned to the variables by calling the `value()` member method of `DialogField`. If a dialog is used within the `RunBase` framework, as it is used in this example, the best place to assign dialog control values to variables is the `getFormDialog()` member method. The `RunBase` framework calls this method right after the user clicks on **OK**.

The main processing is done in the `run()` method. For demonstration purposes, this class only shows the user input in the **Infolog** tab on top of the browser window.

In order to make this class runnable, the `main()` static method has to be created. Here, we create a new `CustCreate` object and invoke the user dialog by calling the `prompt()` method. Once the user has finished entering customer details by clicking on **OK**, we call the `run()` method to process the data.

Handling the dialog event

Sometimes, in the user interface, it is necessary to change the status of one field depending on the status of another field. For example, if the user marks the **Show filter** checkbox, then another field, **Filter**, appears or becomes enabled. In AOT forms, this can be done using the `modified()` input control event. However, if this feature is required on runtime dialogs, handling events is not that straightforward.

Often, existing dialogs have to be modified in order to support events. The easiest way to do this is, of course, to convert a dialog into an AOT form. However, when the existing dialog is complex enough, a more cost-effective solution would probably be to implement dialog event handling instead of converting into an AOT form. Event handling in dialogs is not flexible, as in the case of AOT forms; but in most cases, it does the job.

In this recipe, we will create a dialog similar to the previous dialog, but instead of entering the customer number, we will be able to select the number from a list. Once the customer is selected, the rest of the fields will be filled in automatically by the system from the customer record.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Add a new class named `MyDialogSelect` with the following code snippet:

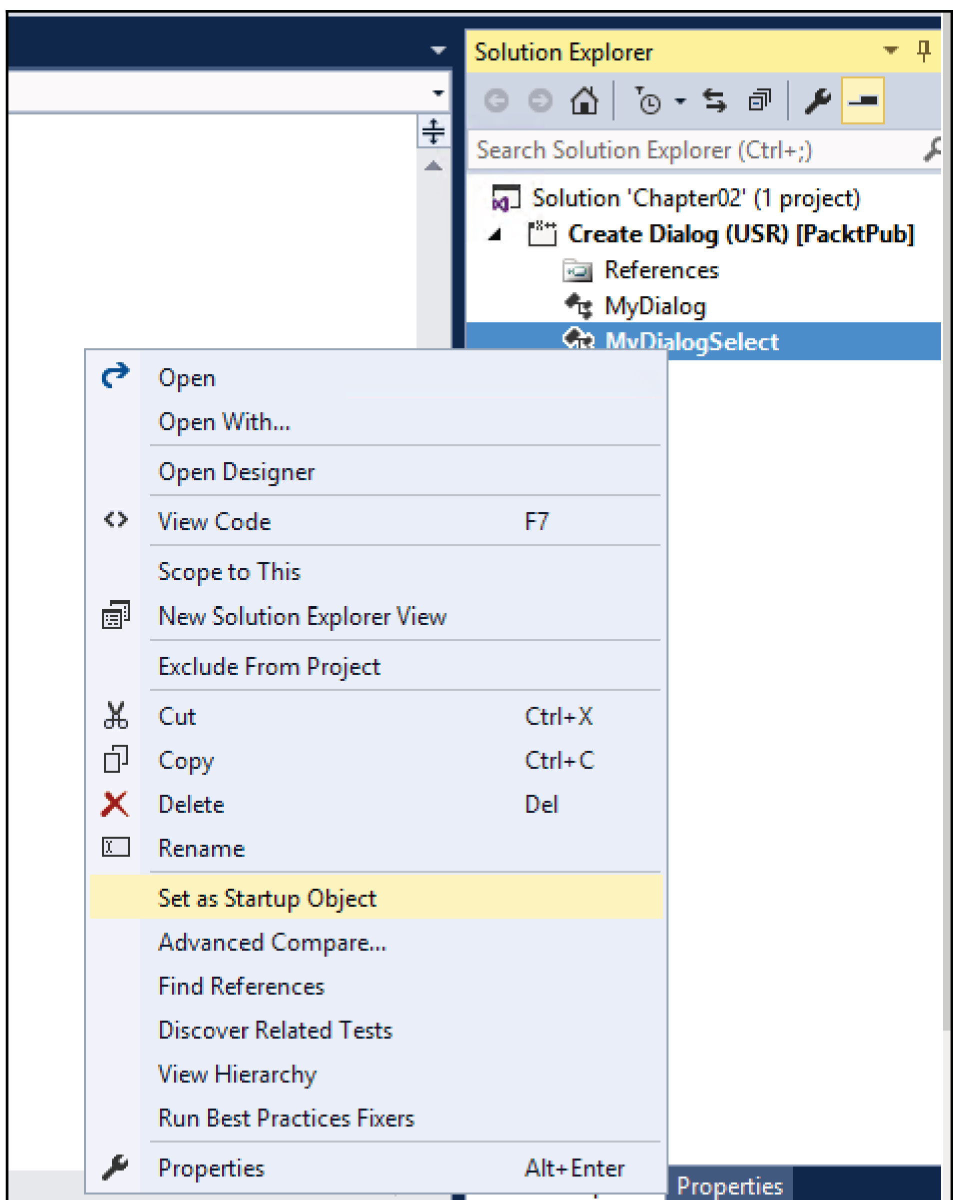
```
class MyDialogSelect extends RunBase
{
   DialogField fieldAccount;
   DialogField fieldName;
   DialogField fieldGroup;
   DialogField fieldCurrency;
   DialogField fieldPaymTermId;
   DialogField fieldPaymMode;


  public container pack()
  {
    return conNull();
  }

  public boolean unpack(container _packedClass)
  {
    return true;
  }
```

2. Create a dialog method to capture run time user inputs for customer details:

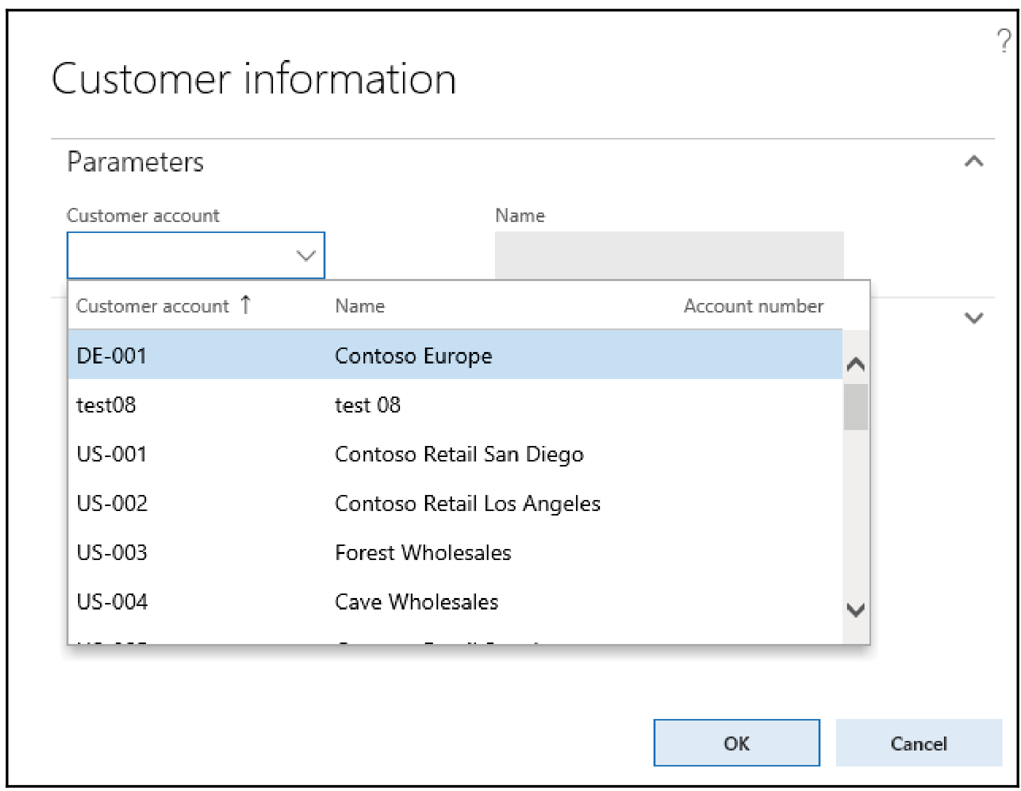
```
Object dialog()
{
   Dialog          dialog;
   DialogGroup     groupCustomer;
   DialogGroup     groupPayment;

   dialog = super();

   dialog.caption("Customer information");
   dialog.allowUpdateOnSelectCtrl(true);

   fieldAccount   = dialog.addField
   (extendedTypeStr(CustAccount),"Customer account");

   fieldName   =dialog.addField
   (extendedTypeStr(CustName));
   fieldName.enabled(false);

   dialog.addTabPage("Details");
```

```
        groupCustomer   = dialog.addGroup("Setup");
        fieldGroup   = dialog.addField
        (extendedTypeStr(CustGroupId));
        fieldCurrency = dialog.addField
        (extendedTypeStr(CurrencyCode));
        fieldGroup.enabled(false);
        fieldCurrency.enabled(false);

        groupPayment    = dialog.addGroup("Payment");
        fieldPaymTermId =dialog.addField
        (extendedTypeStr(CustPaymTermId));
         fieldPaymMode = dialog.addField
         (extendedTypeStr(CustPaymMode));
        fieldPaymTermId.enabled(false);
        fieldPaymMode.enabled(false);

          return dialog;
    }

    public void dialogSelectCtrl()
    {
        CustTable custTable;

        custTable = CustTable::find(fieldAccount.value());
        fieldName.value(custTable.name());
        fieldGroup.value(custTable.CustGroup);
        fieldCurrency.value(custTable.Currency);
        fieldPaymTermId.value(custTable.PaymTermId);
        fieldPaymMode.value(custTable.PaymMode);
    }

    public static void main(Args _args)
    {
        MyDialogSelect myDialogSelect = new MyDialogSelect();

        if (myDialogSelect.prompt())
         {
            myDialogSelect.run();
         }
    }

}
```

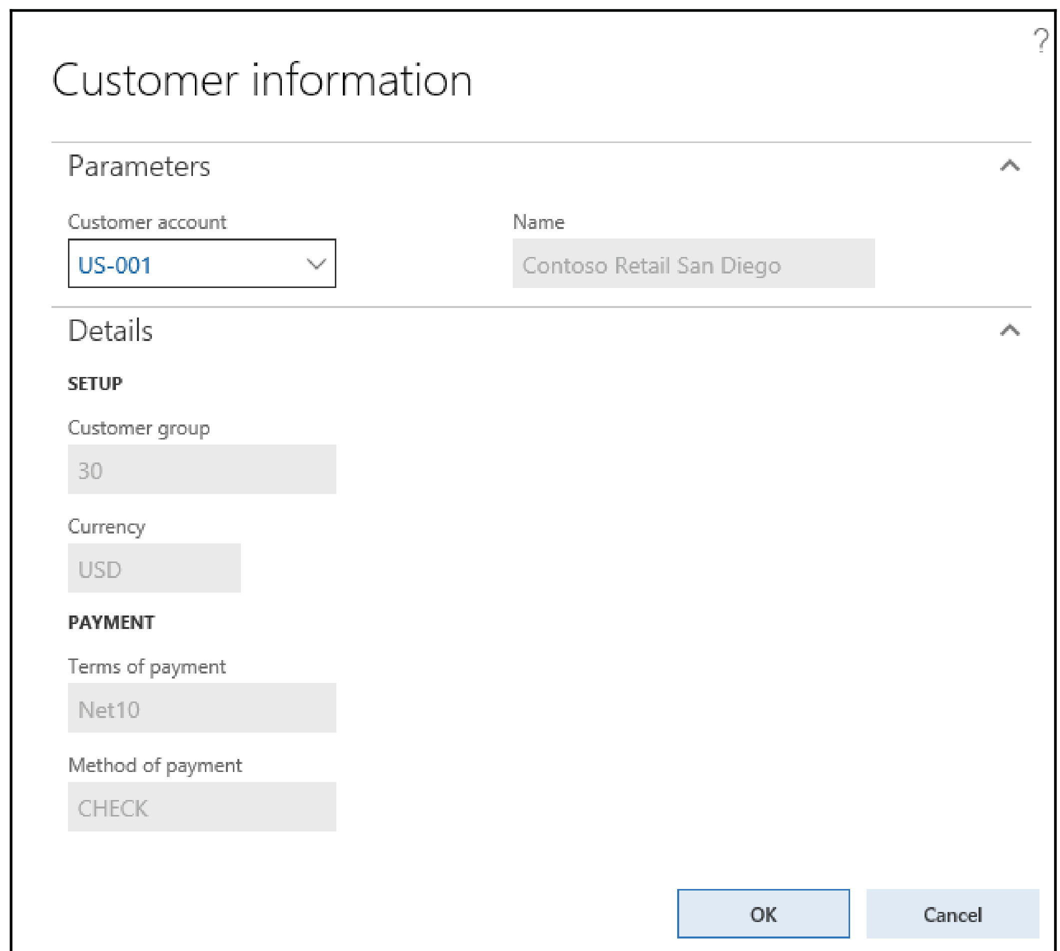
3. Set this class as **Set as Startup Object**



4. Save all your changes and build your project. Now run the project. The following form will appear in an internet browser.
5. Run the project, select any customer from the list, and move the cursor to the next control. Notice how the rest of the fields were automatically populated with the customer's information, as shown in the following screenshot:



6. When you click on the **Details** tab page, you will see more information about the customer, as shown in the following screenshot:



How it works...

The new class named `MyDialogSelect` is actually a copy of the `MyDialog` class from the previous recipe, with a few changes. In its class declaration, we leave all the `DialogField` declarations and remove the rest of the variables.

The `pack()` and `unpack()` methods remain the same, as we are not using any of their features.

In the `dialog()` member method, we call the `allowUpdateOnSelectCtrl()` method with the `true` argument to enable input control event handling. We also disable all the controls, apart from **Customer account,** by calling `enable()` with the `false` parameter for each control.

The `dialogSelectCtrl()` member method of the `RunBase` class is called every time the user modifies any input control in the dialog. It is the place where we have to add all the required code to ensure that in our case, all the controls are populated with the correct data from the customer record-once **Customer account** is selected.

The `main()` method ensures that the class is runnable.

See also

- The *Creating dialogs using the RunBase framework* recipe

Creating dialogs using the SysOperation framework

SysOperation is a framework in Dynamics 365 for Finance and Operations that allows application logic to be written in a way that supports running operations interactively or via the D365 batch server. The SysOperation framework follows the **MVC** (**Model-View-Controller**) pattern. As the name implies, the MVC pattern isolates the Model, View, and Controller components, which makes the process loosely coupled built over the SysOperation framework. Depending on parameters, the controller can execute different service operations under four main execution modes. Regardless of which mode a service is running in, the code runs on a server. This makes the minimum number of round trips between server and client.

- **Synchronous**: When a service is run in synchronous mode, although it runs on a server, it freezes the Dynamics 365 for Operations browser client. A call is initiated from the client and an object is marshaled to the server to run in CIL. This is good for smaller processes.

- **Asynchronous**: In an asynchronous call to service, the client remains responsive. They only work using the WCF asynchronous service call mechanism. This is why it is necessary to have it running as an AIF service. One should drop it to the Dynamics 365 for Operations service group and redeploy the service group. This is good for lengthy processes where durability is not important.
- **Reliable Asynchronous**: Works like batch service. As soon as a call is initiated to run a service in reliable asynchronous mode, it is scheduled to be run on the batch server instantly, but removed as soon as it finishes the job. One can see it among other scheduled jobs. Since it runs on a batch server, it can exploit the power of parallel processing. It is used in scenarios where a job needs to be run on a server and not to schedule. There is room for performance enhancement making use of parallel processing among different AOS.
- **Scheduled Batch**: A job is scheduled to run on a batch server. This is similar to reliable asynchronous, except that it does not delete the job instance once the job is finished. This is used for jobs that need to be run repeatedly at specified time intervals. There is room for performance enhancements making use of parallel processing among different AOS.

In this recipe, we will create a dialog which will take certain parameters. Based on the parameters provided, customer's balance will be displayed onscreen by pressing the button on the All Customers form to Display balances. It can be opened by navigating to **Accounts receivable | Customers | All Customers**.

Getting ready

We will be using the following development artifacts for demonstration purposes.

- **Data contract**: The data contract (`CustBalanceDataContract`) is the model class in which we define which attributes we need for our operation, commonly set as parameters by the user in a dialog. It's just a model class with an attribute, in which we will use the `DataContractAttribute` attribute to decorate our class declaration. For each member variable, we have to define one parm methods using the attribute `DataMemberAttribute`, which will work like getter setter method. Additionally, if we want some more methods to be available to us, we can also extend the standard class `SysOperationDataContractBase`. With this class, we can define how our basic dialog will look to the user. We can define our labels, groups, sizes, and types of parameters.

How to do it...

Carry out the following steps in order to complete this recipe:

1. In the VS project , create a new class called `CustBalanceDataContract` with the following code snippet:

```
[
DataContractAttribute,
SysOperationContractProcessingAttribute
(classStr(CustBalanceUIBuilder)),
SysOperationGroupAttribute
('Date',"@ApplicationPlatform:SingleSpace", '1')
]
class CustBalanceDataContract implementsSysOperationValidatable
{
   NoYesId     allowModifyDate;
   TransDate   transDate;
   str packedQuery;

   /// <summary>
   /// Gets or sets the value of the datacontract parameter
        DateTransactionDate.
   /// </summary>
   /// <param name="_transDate">
   /// The new value of the datacontract parameter
        DateTransactionDate;
   /// </param>
   /// <returns>
   ///  The current value of datacontract parameter
         DateTransactionDate
   /// </returns>
   [DataMemberAttribute('DateTransactionDate')
    ,SysOperationLabelAttribute(literalStr("@SYS11284")),
     SysOperationGroupMemberAttribute('Date'),
     SysOperationDisplayOrderAttribute('1')] // today's date
   public TransDate parmTransDate
    (TransDate _transDate = transDate)
    {
       transDate = _transDate;

       return transDate;
    }

   /// <summary>
   /// Gets or sets the value of the datacontract parameter
        DateControl.
```

```
/// </summary>
/// <param name="_allowModifyDate">
/// The new value of the datacontract parameter
     DateControl;
/// </param>
/// <returns>
///  The current value of datacontract parameter
       DateControl
/// </returns>
[DataMemberAttribute('DateControl'),
SysOperationLabelAttribute("Enable date control"),
SysOperationGroupMemberAttribute('Date'),
SysOperationDisplayOrderAttribute('0')]
public NoYesId parmAllowModifyDate
 (NoYesId _allowModifyDate   = allowModifyDate)
 {
   allowModifyDate = _allowModifyDate;
   return allowModifyDate;
 }

/// <summary>
/// Validates the dialog values for errors.
/// </summary>
/// <returns>
/// false if an error has occurred in the dialog values;
    otherwise, true .
/// </returns>
/// <remarks>
/// The dialog values are handled through the contract.
/// </remarks>
public boolean validate()
 {
   boolean ret = true;

   if(!transDate && allowModifyDate)
   ret = checkFailed('Transaction date cannot be empty');

   return ret;
 }

[DataMemberAttribute,
AifQueryTypeAttribute
 ('_packedQuery',   querystr(CustTableSRS))
]
public str parmQuery(str _packedQuery = packedQuery)
 {
   packedQuery = _packedQuery;
   return packedQuery;
```

```
    }

    public Query getQuery()
    {
        return new
        Query(SysOperationHelper::base64Decode(packedQuery));
    }

    public void setQuery(Query _query)
    {
        packedQuery
         =SysOperationHelper::base64Encode(_query.pack());
    }

}
```

Here, `SysOperationGroupAttribute` specifies how we group the contract parameters and provides the order in which to display the group. Data contract also implements the `SysOperationValidatable` interface, due to which we need to override the `Validate()` method and validate parameters before actual execution begins. Using `SysOperationContractProcessingAttribute`, we specify the `UIbuilder` class to modify the parameter's behavior at runtime. We will create this UI builder class later in this chapter.

2. In the VS project , create a new class called `CustBalanceController` with the following code snippet:

- **Controller**: As the name implies, this class has great responsibility for initiating the operation. This class holds all the information regarding execution mode; it should show a progress form or dialog. It is best practice not to write the whole business login in the `Controller` class itself. That's why, in this demo, we have created one service class to write our business logic, and that service class reference is provided in this controller class main method.

```
class CustBalanceController extends
 SysOperationServiceController
{
    str packedQuery;
    CustBalanceDataContract contract;

   /// <summary>
   /// Sets the query ranges based on caller.
   /// </summary>
   /// <param name="_query">
   /// The hold the <c>Query</c> object of the service.
```

```
/// </param>
public void setRanges()
{
    QueryBuildRange         queryBuildRange;
    QueryBuildDataSource    queryBuildDataSource;
    FormDataSource          custTableDS;
    CustTable               custTable;
    str                     range;
    Query                   _query;

   contract = this.getDataContractObject() as
    CustBalanceDataContract;
   _query   = contract.getQuery();
   if (this.parmArgs()
   &&  this.parmArgs().caller()
   &&  this.parmArgs().dataset() == tableNum(CustTable))
    {
       custTableDS = FormDataUtil::getFormDataSource
        (this.parmArgs().record());

         if (_query && custTableDS)
        {
           // build range
           for (custTable = custTableDS.getFirst(true) ?
           custTableDS.getFirst(true): custTableDS.cursor();
           custTable;
           custTable = custTableDS.getNext())
           {
              range = range == '' ? custTable.AccountNum :
              range
              + ',' + custTable.AccountNum;
           }

               if (range)
            {
               queryBuildDataSource =
                _query.dataSourceTable(tableNum(CustTable));

               // check for QueryBuildDataSource
               if (queryBuildDataSource)
              {
                 // clear the old range,and then add it
                   queryBuildDataSource.clearRanges();
               if (!queryBuildRange)
               {
                  queryBuildRange
                  =queryBuildDataSource.addRange
                   (fieldNum(CustTable, AccountNum));
```

```
                    }
                    queryBuildRange.value(range);
                   }
                }
            }
        }
            contract .setQuery(_query);
    }

        public static void main(Args _args)
        {
            CustBalanceController   controller = new
            CustBalanceController(classStr(CustBalanceService),
            methodStr(CustBalanceService,processData),
            SysOperationExecutionMode::Synchronous);

            controller.parmArgs(_args);
            controller.setRanges();
            controller.startOperation();
        }

    }
```

Here, we extend the `SysOperationServiceController` class to inherit controller capabilities. The main method is used to create an instance of the controller class, where we specify the service class and service method which need to be called to execute the business logic. The `setRanges()` method is called to specify ranges based on the caller.

- **Service**: As I mentioned earlier, it's not a good practice to keep the whole business logic in one controller class, because it would be a big responsibility for a single class to handle. That's why, here, we have created a `Service` class which is referenced in the `Controller` class.

3. In the VS project , create a new class called `CustBalanceController` with the following code snippet:

```
class CustBalanceService
{
   [SysEntryPointAttribute]
    public void processData(CustBalanceDataContract
    _custBalanceDataContract)
    {
        QueryRun    queryRun;
        CustTable   custTable;
        Amount      balance;
```

```
        ;
        // create a new queryrun object
        queryRun = new queryRun
        (_custBalanceDataContract.getQuery());

        // loop all results from the query
        while(queryRun.next())
       {
          custTable = queryRun.get(tableNum(custTable));

          if(_custBalanceDataContract.parmTransDate())
          balance =  custTable.balanceMST
           (dateNull(),
            _custBalanceDataContract.parmTransDate());
        else
           balance = custTable.balanceMST();
           // display the balance
           info(strFmt("%1 - %2",custTable.AccountNum,balance));
       }
    }
}
```

Here, we get the contract parameters and execute the business logic. The customer balance in the accounting currency is displayed as at a date if a certain date is specified. Herein, we could also multithread our process.

- **UIBuider**: This class is only required when you want to play with added parameters (data member attributes) in the contract class. For example, modifying lookup or enabling/disabling certain parameters on a dialog.

4. In the VS project , create a new class called `CustBalanceUIBuilder` with the following code snippet:

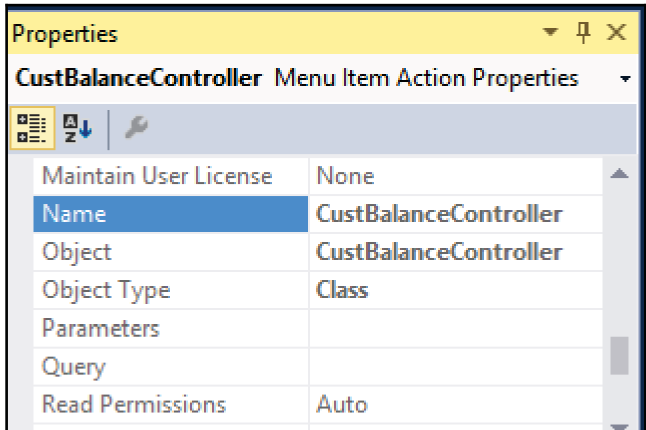
```
class CustBalanceUIBuilder extends
SysOperationAutomaticUIBuilder
{
   DialogField     dialogFieldAllowModifyDate;
   DialogField     dialogFieldTransDate;

   CustBalanceDataContract custBalanceDataContract;

   public boolean allowModifyDateModified(FormCheckBoxControl
    _checkBoxControl)
    {
       // set enabled or disabled based on checkbox
       dialogFieldTransDate.enabled
        (any2enum(dialogFieldAllowModifyDate.value()));
```

```
            // or alternatively
            //
            dialogFieldTransDate.enabled
             (_checkBoxControl.checked());
            return true;
        }

        public void postBuild()
        {
            ;
            super();

            // get datacontract
            custBalanceDataContract = this.dataContractObject();

            // get dialog fields
            dialogFieldTransDate= this.bindInfo().getDialogField
             (custBalanceDataContract,methodstr
              (custBalanceDataContract,parmTransDate));
            dialogFieldAllowModifyDate=
            this.bindInfo().getDialogField
             (custBalanceDataContract, methodstr
              (custBalanceDataContract,parmAllowModifyDate));

            // register override methods
            dialogFieldAllowModifyDate.registerOverrideMethod
            (methodstr(FormCheckBoxControl, modified),
            methodstr(CustBalanceUIBuilder,
            allowModifyDateModified), this);
            dialogFieldTransDate.enabled
            (any2enum(dialogFieldAllowModifyDate.value()));
        }
    }
```

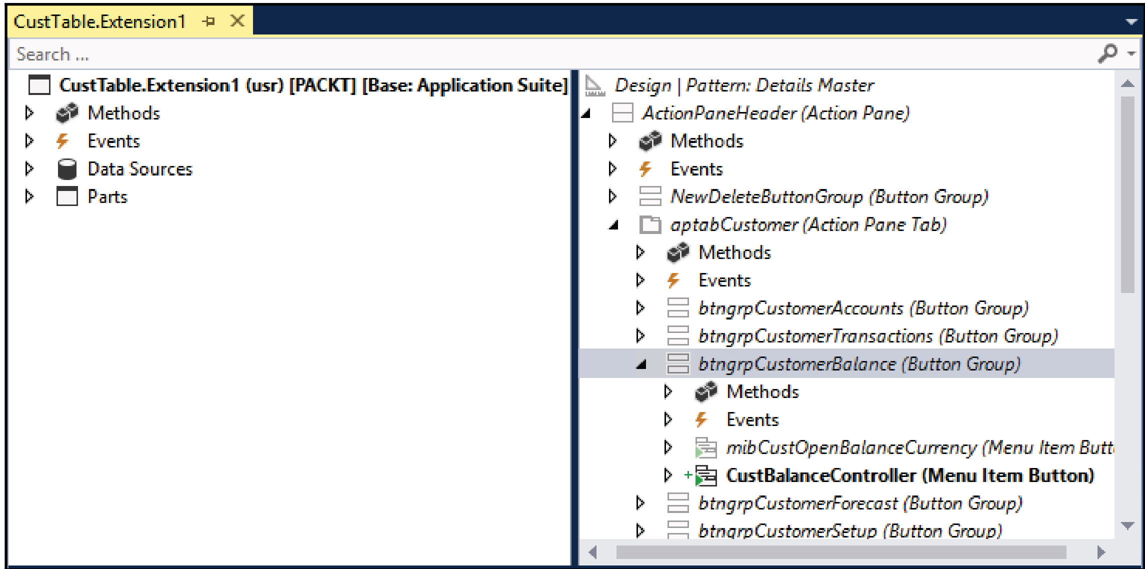
Here, we override the `postBuild` method and get the two dialog fields. Taking it further, we register the `allowModifyDateModified()` on event modified of our `dialogFieldAllowModifyDate` control.

Finally, we need to create an action menu item as an entry point to execute the preceding code:

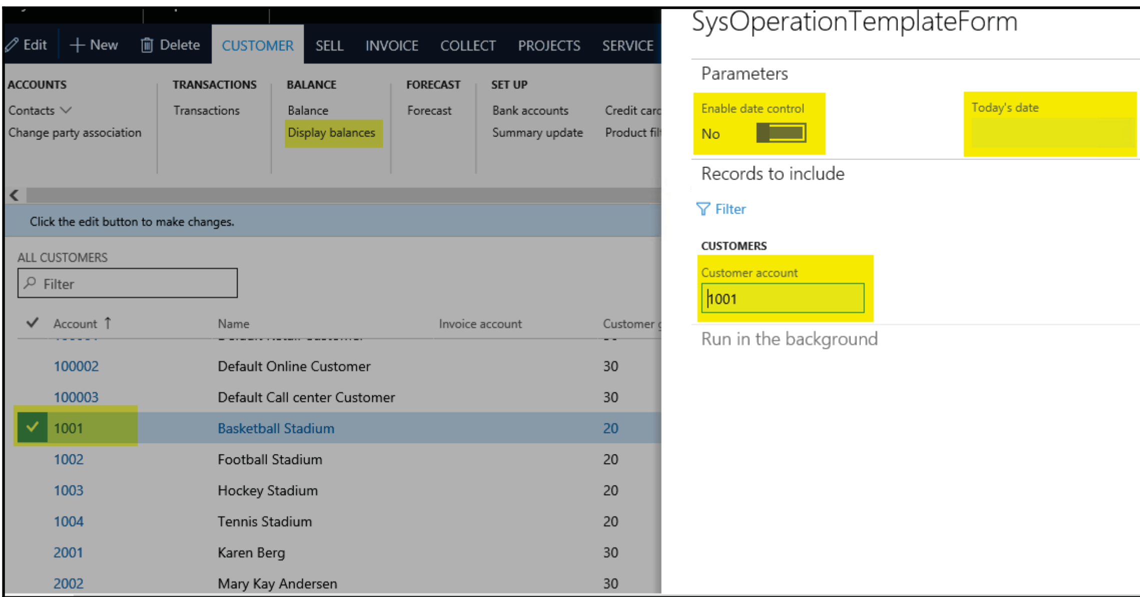
1. In the VS project, create a new action menu item called `CustBalanceController` with the following properties:



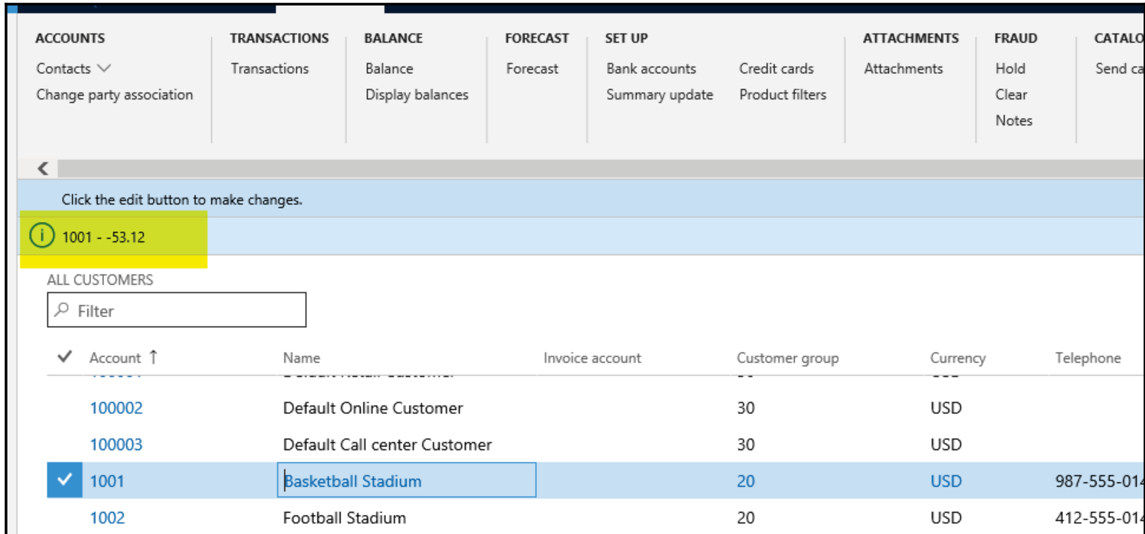
2. Place the menu item at **Accounts receivable** | **Customers** | **All Customers** | **Customer** | **Balance** | **Display balance,** as shown in the following screenshot:



3. Finally, our customer form will look as follows:



4. The final output will look as follows:



Building a dynamic form

A standard approach to creating forms in Dynamics 365 for Finance and Operations is to build and store `form` objects in the AOT. It is possible to achieve a high level of complexity using this approach. However, in a number of cases, it is necessary to have forms created dynamically. In a standard Dynamics 365 for Finance and Operations application, we can see that application objects, such as the **Table browser** form, various lookups, or dialogs, are built dynamically. Even in Dynamics 365 for Finance and Operations, where we have a browser-based interface, every form or dialog opens in a browser only.

In this recipe, we will create a dynamic form. In order to show how flexible the form can be, we will replicate the layout of the existing **Customer groups** form located in the **Accounts receivable** module. The Customers form can be opened by navigating to **Accounts receivable** | **Setup** | **Customers**.

How to do it...

Carry out the following steps in order to complete this recipe:

1. In the AOT, create a new class called `CustGroupDynamicForm` with the following code snippet.
2. We will run this class directly to get output. So, just for ease, we will write all code in the main method of this class:

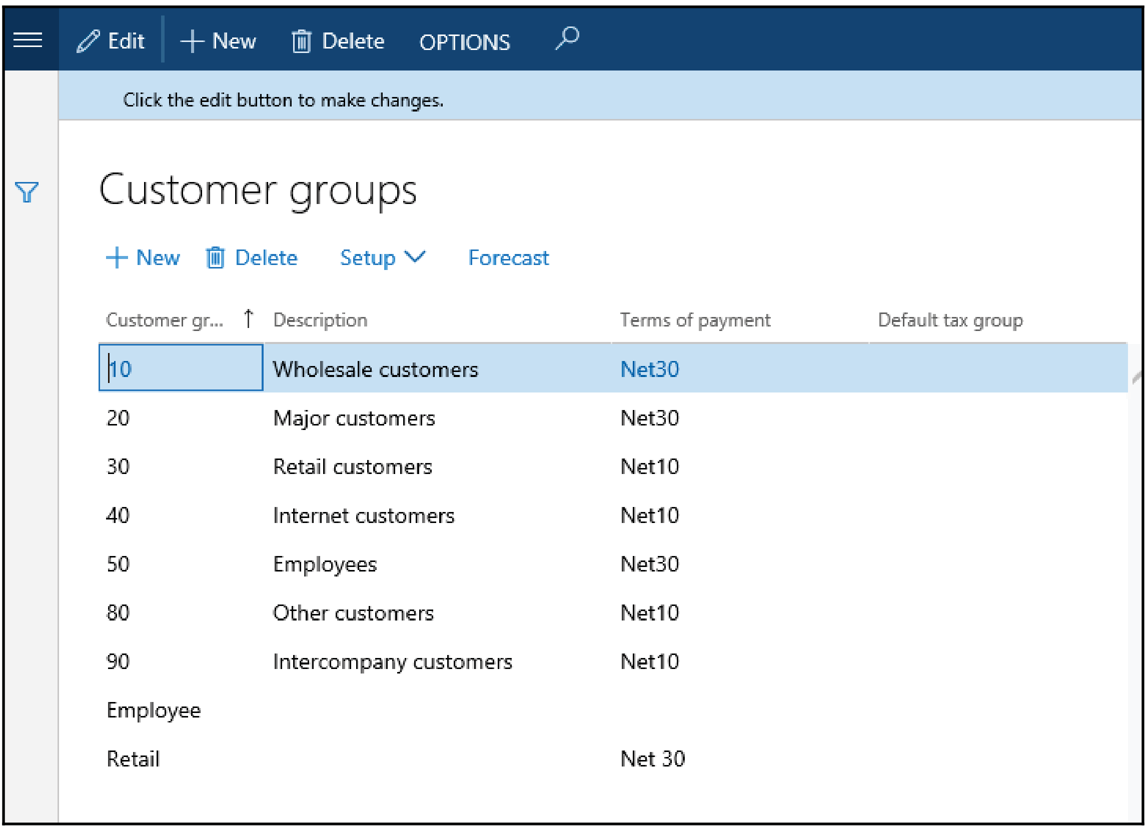
```
class CustGroupDynamicForm
{
   public static void main(Args _args)
  {
     // Object declarations
     DictTable                         dictTable;
     Form                              form;
     FormBuildDesign                   design;
     FormBuildDataSource               ds;
     FormBuildActionPaneControl        actionPane;
     FormBuildActionPaneTabControl     actionPaneTab;
     FormBuildButtonGroupControl       btngrp1;
     FormBuildButtonGroupControl       btngrp2;
     FormBuildCommandButtonControl     cmdNew;
     FormBuildCommandButtonControl     cmdDel;
     FormBuildMenuButtonControl        mbPosting;
     FormBuildFunctionButtonControl    mibPosting;
     FormBuildFunctionButtonControl    mibForecast;
     FormBuildGridControl              grid;
```

```
FormBuildGroupControl           grpBody;
Args                            args;
FormRun                         formRun;
#Task

dictTable = new DictTable(tableNum(CustGroup));

// Use Form class to create a dynamics form and
//use its method to set different properties.
form = new Form();
form.name("CustGroupDynamic");

//Add datasource in Form
ds = form.addDataSource(dictTable.name());
ds.table(dictTable.id());

//Set Design prperties
design = form.addDesign('Design');
design.caption("Customer groups");
design.style(FormStyle::SimpleList);
design.titleDatasource(ds.id());

//Add ActionPan design controls and set their
//properties
actionPane = design.addControl(
FormControlType::ActionPane, 'ActionPane');
actionPane.style(ActionPaneStyle::Strip);
actionPaneTab = actionPane.addControl(
FormControlType::ActionPaneTab, 'ActionPaneTab');
btngrp1 = actionPaneTab.addControl(
FormControlType::ButtonGroup, 'NewDeleteGroup');
btngrp2 = actionPaneTab.addControl(
FormControlType::ButtonGroup, 'ButtonGroup');

//Add CommandButton design controls and set their
//properties

cmdNew = btngrp1.addControl(
FormControlType::CommandButton, 'NewButton');
cmdNew.primary(NoYes::Yes);
cmdNew.command(#taskNew);

//Add CommandButton design controls and set their
//properties

cmdDel = btngrp1.addControl(
FormControlType::CommandButton, 'DeleteButton');
cmdDel.text("Delete");
```

```
        cmdDel.saveRecord(NoYes::Yes);
        cmdDel.primary(NoYes::Yes);
        cmdDel.command(#taskDeleteRecord);

        //Add MenuButton design controls and set their
        //properties

        mbPosting = btngrp2.addControl(
        FormControlType::MenuButton, 'MenuButtonPosting');
        mbPosting.helpText("Set up related data for the group.");
        mbPosting.text("Setup");

        mibPosting = mbPosting.addControl(
        FormControlType::MenuFunctionButton, 'Posting');
        mibPosting.text('Item posting');
        mibPosting.saveRecord(NoYes::No);
        mibPosting.dataSource(ds.id());
        mibPosting.menuItemName
        (menuitemDisplayStr(InventPosting));

        mibForecast = btngrp2.addControl(
        FormControlType::MenuFunctionButton, 'SalesForecast');
        mibForecast.text('Forecast');
        mibForecast.saveRecord(NoYes::No);
        mibForecast.menuItemName(
        menuitemDisplayStr(ForecastSalesGroup));

        //Add Grid design controls and set their
        //properties

        grpBody = design.addControl(FormControlType::Group,
'Body');
        grpBody.heightMode(FormHeight::ColumnHeight);
        grpBody.columnspace(0);
        grpBody.style(GroupStyle::BorderlessGridContainer);

        grid = grpBody.addControl(FormControlType::Grid, "Grid");
        grid.dataSource(ds.name());
        grid.showRowLabels(false);
        grid.widthMode(FormWidth::ColumnWidth);
        grid.heightMode(FormHeight::ColumnHeight);


        //Add fields in Grid and set their        //properties

        grid.addDataField
        (ds.id(), fieldNum(CustGroup,CustGroup));
```

```
        grid.addDataField(
        ds.id(), fieldNum(CustGroup,Name));

        grid.addDataField(
        ds.id(), fieldNum(CustGroup,PaymTermId));

        grid.addDataField(
        ds.id(), fieldnum(CustGroup,ClearingPeriod));

        grid.addDataField(
        ds.id(), fieldNum(CustGroup,BankCustPaymIdTable));

        grid.addDataField(
        ds.id(), fieldNum(CustGroup,TaxGroupId));
        args = new Args();
        args.object(form);

        formRun = classFactory.formRunClass(args);
        formRun.init();
        formRun.run();

        formRun.detach();
    }

}
```

3. In order to test the form, run the `CustGroupDynamic` class. Notice that the form is similar to the one located in **Accounts receivable**, which can be obtained by navigating to **Setup** | **Customers** | **Customer groups**, as shown in the following screenshot:



How it works...

We start the code by declaring variables. Note that most of the variable types begin with `FormBuild`, which are a part of a set of application classes used to build dynamic forms. Each of these types corresponds to the control types that are manually used when building forms in the AOT.

Right after the variable declaration, we create a `dictTable` object based on the `CustGroup` table. We will use this object several times later in the code. Then, we create a `form` object and set a name by calling the following lines of code:

```
form = new Form();
form.name("CustGroupDynamic");
```

The name of the `form` object is not important, as this is a dynamic form. The form should have a data source, so we add one by calling the `addDataSource()` method to the `form` object and by providing a previously created `dictTable` object, as shown here:

```
ds = form.addDataSource(dictTable.name());
ds.table(dictTable.id());
```

Every form has a design, so we add a new design, define its style as a simple list, and set its `title` data source, as shown in the following code snippet:

```
design = form.addDesign('Design');
design.caption("Customer groups");
design.style(FormStyle::SimpleList);
design.titleDatasource(ds.id());
```

Once the design is ready, we can start adding controls from the code as if we were doing this from the AOT. The first thing you need to do is to add a `strip` action pane with its buttons:

```
actionPane = design.addControl(
 (FormControlType::ActionPane, 'ActionPane');
actionPane.style(ActionPaneStyle::Strip);
actionPaneTab = actionPane.addControl(
 (FormControlType::ActionPaneTab, 'ActionPaneTab');
btngrp1 = actionPaneTab.addControl(
```

Right after the action pane, we add an automatically expanding grid that points to the previously mentioned data source. Just to follow best practices, we place the grid inside a `Group` control:

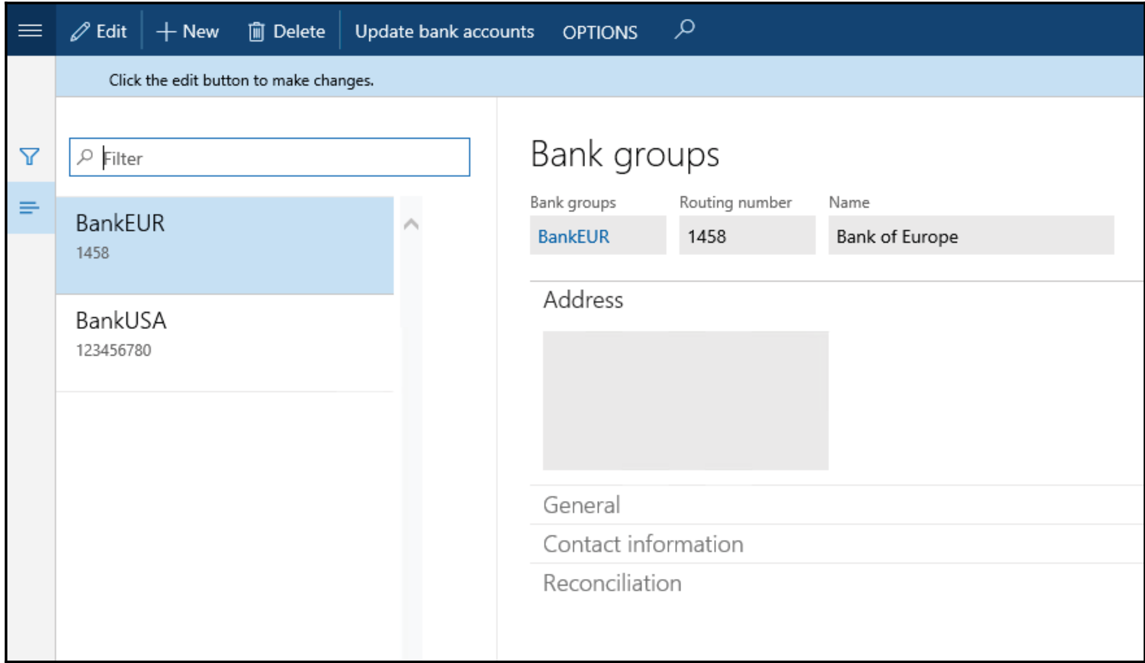
```
grpBody = design.addControl(FormControlType::Group, 'Body');
grpBody.heightMode(FormHeight::ColumnHeight);
grpBody.columnspace(0);
grpBody.style(GroupStyle::BorderlessGridContainer);

grid = grpBody.addControl(FormControlType::Grid, "Grid");
grid.dataSource(ds.name());
grid.showRowLabels(false);
grid.widthMode(FormWidth::ColumnWidth);
grid.heightMode(FormHeight::ColumnHeight);
```

Next, we add a number of grid controls that point to the relevant data source fields by calling `addDataField()` on the `grid` object. The last thing is to initialize and run the form. Here, we use the recommended approach to creating and running forms using the globally available `classFactory` object.

Adding a form splitter

In Dynamics 365 for Finance and Operations, complex forms consist of one or more sections. Each section may contain grids, groups, or any other element. In order to maintain section sizes while resizing the form, the sections are normally separated by so-called **splitters**. Splitters are not special Dynamics 365 for Finance and Operations controls; they are `Group` controls with their properties modified so that they look like splitters. Most of the multisection forms in Dynamics 365 for Finance and Operations already contain splitters.

In this recipe, in order to demonstrate the usage of splitters, we will modify one of the existing forms that does not have a splitter. We will modify the **Account reconciliation** form in the **Cash and bank management** module. You can open this module by navigating to **Cash and bank management** | **Setup** | **Bank group**. From the following screenshot, you can see that it is not possible to control the size of each grid individually and that they are resized automatically using a fixed radio button when resizing the form:



In this recipe, we will demonstrate the usage of splitters by improving this situation. We will add a form splitter between two grids in the mentioned form. This will allow users to define the sizes of both grids in order to ensure that the data is displayed optimally.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Add the `BankGroup` form in the AOT and, in the form's design, add a new `Group` control right after the `ActionPane` control with the following properties:

Property	Value
Name	Top
AutoDeclaration	Yes
FrameType	None

2. Move the `DetailsHeader` and `Tab` controls into the newly created group.
3. Change the following properties of the existing `DetailsHeader` group:

Property	Value
Top	Auto
Height	Column height

4. Add a new `Group` control immediately below the `Top` group with the following properties:

Property	Value
Name	Splitter
Style	SplitterVerticalContainer
AutoDeclaration	Yes

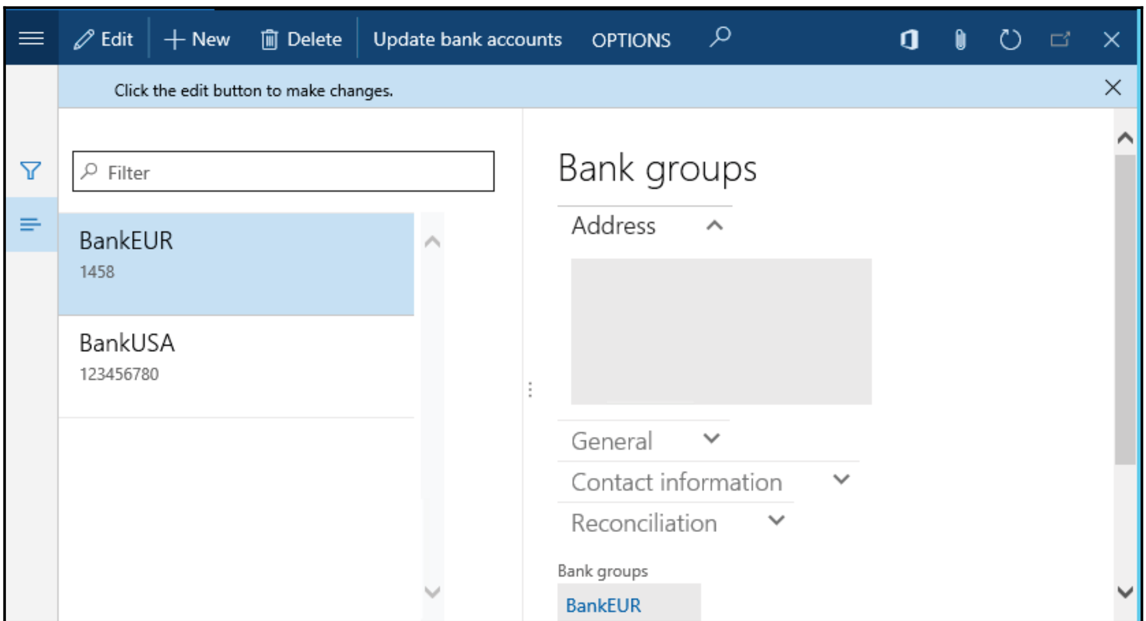
5. Add the following line of code at the bottom of the form's class declaration:

```
SysFormSplitter_Y formSplitter;
```

6. Add the following line of code at the bottom of the form's `init()` method:

```
formSplitter = new SysFormSplitter_Y(Splitter, Top, element);
```

7. Save all your code and build the solution.
8. Now, in order to test the results, navigate to **Cash and bank management** | **Setup** | **Bank groups**. Note that, now, the form has a splitter in the middle, which makes the form look better and allows you to resize both grids, as shown in the following screenshot:



How it works...

Normally, a splitter has to be placed between two form groups. In this recipe, to follow this rule, we need to adjust the **BankGroup** form's design. The `DetailsHeader` group and `Tab` controls are moved to a new group called `Top`. We do not want this new group to be visible to the user, so we set `FrameType` to `None`. Setting `AutoDeclaration` to `Yes` allows you to access this object from the code. Finally, we make this group automatically expand in the horizontal direction by setting its `Width` property to `Column width`. At this stage, the visual form layout does not change, but now we have the upper group ready.

We change its `Top` behavior to `Auto` and make it fully expandable in the vertical direction. The `Height` property of the grid inside this group also has to be changed to `Column height` in order to fill all the vertical space.

In the middle of these two groups, we add a splitter. The splitter is nothing but another group which looks like a splitter. We set its `Style` property to `SplitterVerticalContainer`, which makes this control look like a proper form splitter.

Finally, we have to declare and initialize the `SysFormSplitter_Y` application class, which does the rest of the tasks.

In this way, horizontal splitters can be added to any form. Vertical splitters can also be added to forms using a similar approach. For this, we need to use another application class called `SysFormSplitter_X`.

Creating a modal form

Often, people who are not familiar with computers and software tend to get lost among open application windows. The same can be applied to Dynamics 365 for Finance and Operations. Frequently, a user opens a form, clicks a button to open another one, and then goes back to the first one without closing the second form. Sometimes this happens intentionally, sometimes not, but the result is that the second form gets hidden behind the first one and the user starts wondering why it is not possible to close or edit the first form.

Although it is not best practice, sometimes such issues can be easily solved by making the child form a modal window. In other words, the second form always stays on top of the first one until it is closed. In this recipe, we will make a modal window from the **Create sales order** form.

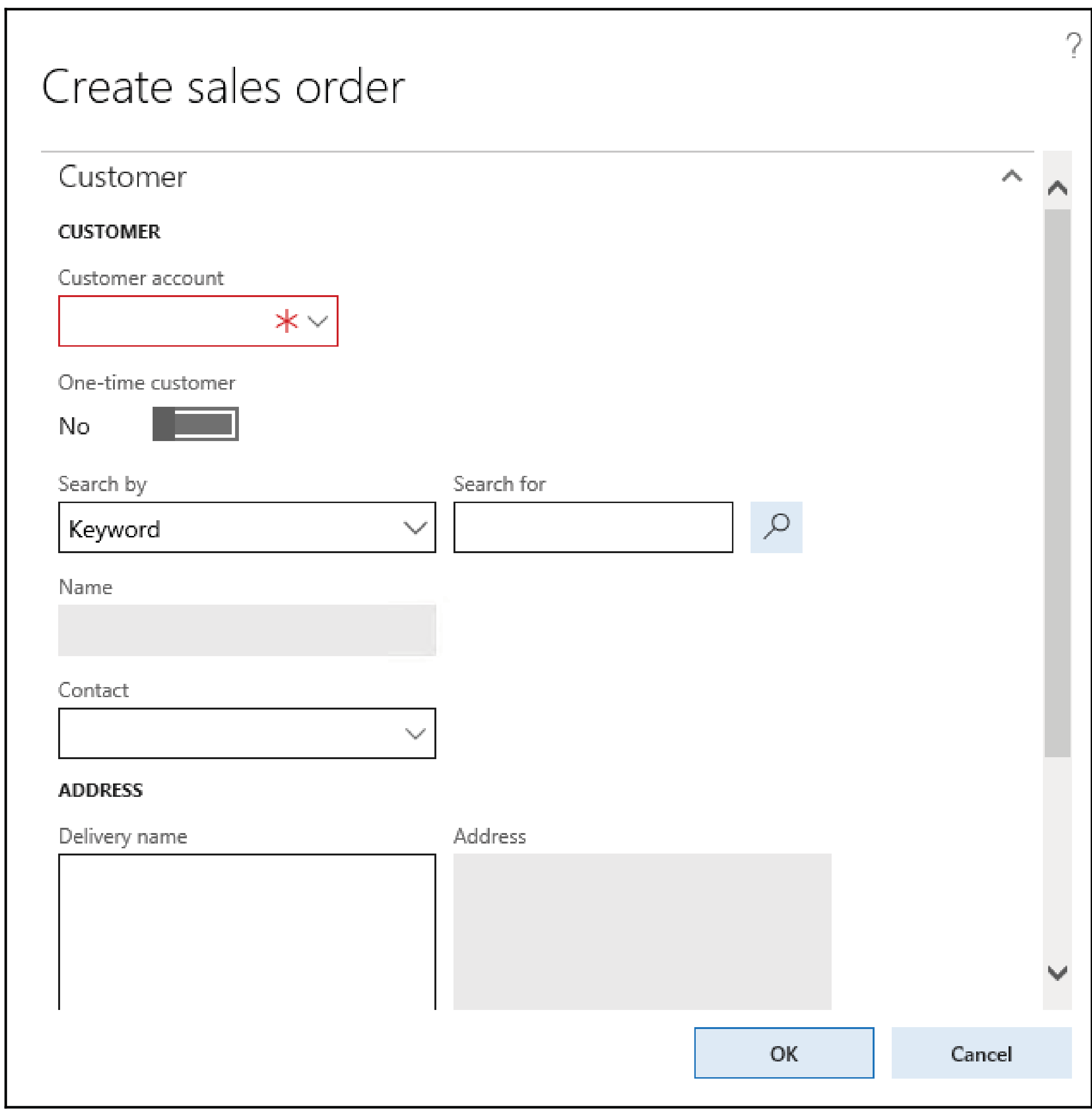
How to do it...

Carry out the following steps in order to complete this recipe:

1. Add the `SalesCreateOrder` form in the project and set its `Design` property:

Property	Value
WindowType	Popup

2. In order to test it, navigate to **Sales and marketing** | **Common** | **Sales orders** | **All sales orders** and start creating a new order. Notice that, now, the **sales order creation** form always stays on top:



How it works...

The form's design has a `WindowType` property, which is set to `Standard` by default. In order to make a form behave as a modal window, we have to change it to `Popup`. Such forms will always stay on top of the parent form.

There's more...

We already know that some of the Dynamics 365 for Finance and Operations forms are created dynamically using the `Dialog` class. If we take a deeper look at the code, we will find that the `Dialog` class actually creates a runtime form. This means that we can apply the same principle--change the relevant form's `design` property. The following lines of code can be added to the `Dialog` object and will do the job:

```
dialog.dialogForm().buildDesign().windowType
  (FormWindowType::Popup);
```

Here, we get a reference to the form's design by first using the `dialogForm()` method of the `Dialog` object to get a reference to the `DialogForm` object, and then we call `buildDesign()` on the latter object. Lastly, we set the `design` property by calling its `windowType()` method with the `FormWindowType::Popup` argument.

See also

- The *Creating dialogs using the RunBase framework* recipe

Modifying multiple forms dynamically

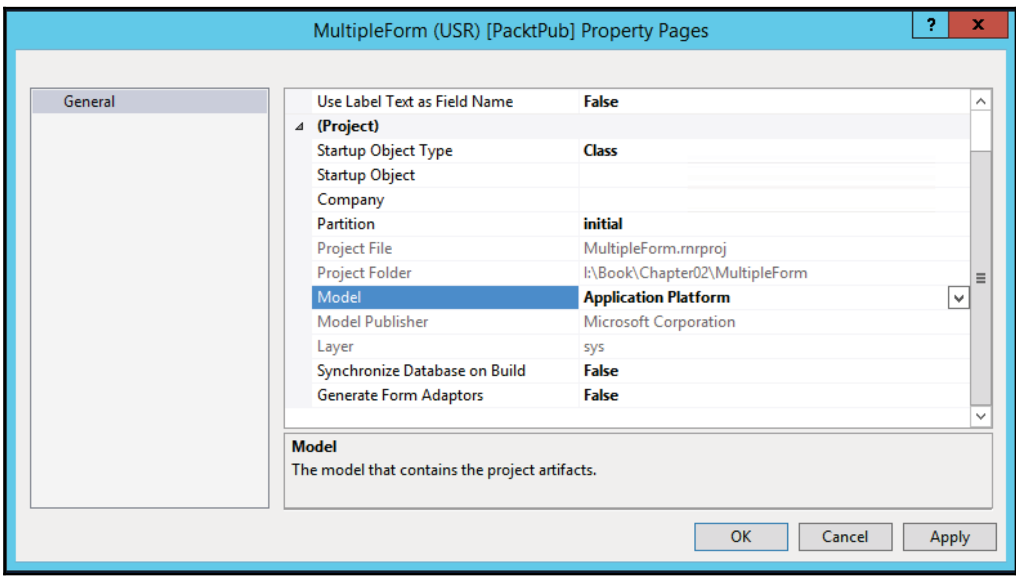
In the standard Dynamics 365 for Finance and Operations, there is a class called `SysSetupFormRun`. The class is called during the run of every form in Dynamics 365 for Operations; therefore, it can be used to override one of the common behaviors for all Dynamics 365 for Finance and Operations forms. For example, different form background colors can be set for different company accounts, some controls can be hidden or added depending on specific circumstances, and so on.

In this recipe, we will modify the `SysSetupFormRun` class to automatically add the **About Dynamics 365 for Operations** button to every form in Dynamics 365 for Finance and Operations.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Add a new project, name it `MultipleForm`, and change the model to Application Platform, as shown in the following screenshot:



2. Add the `FormRun` class and create a new method with the following code snippet:

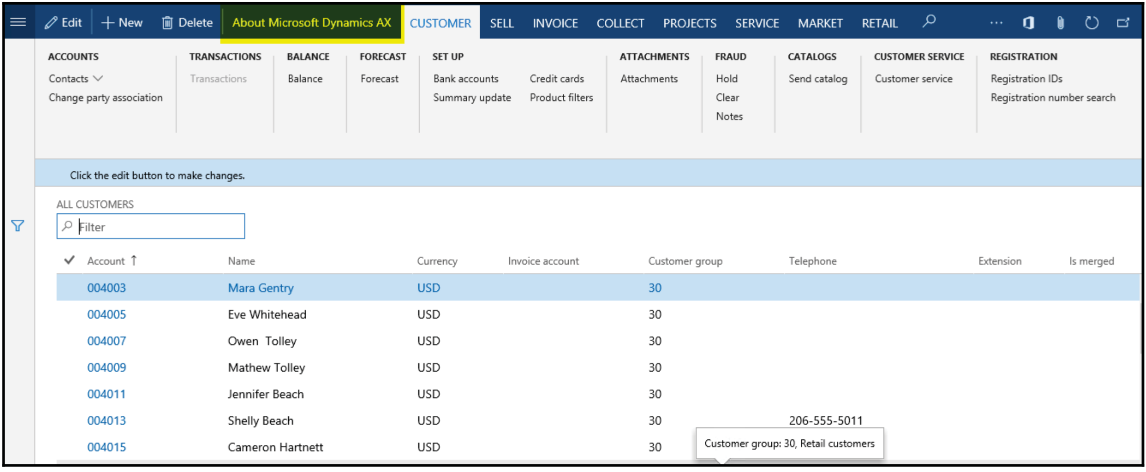
```
private void addAboutButton()
{
   FormActionPaneControl     actionPane;
   FormActionPaneTabControl actionPaneTab;
   FormCommandButtonControl cmdAbout;
   FormButtonGroupControl   btngrp;
   #define.taskAbout(259)

   actionPane = this.design().controlNum(1);
   if (!actionPane ||
      !(actionPane is FormActionPaneControl) ||
      actionPane.style() == ActionPaneStyle::Strip)
    {
      return;
    }

   actionPaneTab = actionPane.controlNum(1);
```

```
        if (!actionPaneTab ||
            !(actionPaneTab is FormActionPaneTabControl))
         {
           return;
         }

        btngrp = actionPaneTab.addControl
         (FormControlType::ButtonGroup, 'ButtonGroup');
        btngrp.caption("About");

        cmdAbout = btngrp.addControl
         (FormControlType::CommandButton, 'About');
        cmdAbout.command(#taskAbout);
        cmdAbout.imageLocation
         (SysImageLocation::EmbeddedResource);
        cmdAbout.normalImage('412');
        cmdAbout.big(NoYes::Yes);
        cmdAbout.saveRecord(NoYes::No);
    }
```

3. In the same class, override its `run()` method with the following code snippet:

```
public void run()
{
  this.addAboutButton();
  super();
}
```

4. In order to test the results, open any list page; for example, go to **Accounts Receivable** | **Customers** | **All customers** and you will notice a new button named **About Dynamics 365 for Operations** in the **Action** pane, as shown in the following screenshot:



How it works...

The `SysSetupFormRun` is the application class that is called by the system every time a user runs a form. The best place to add our custom control is in its `run()` method.

We use the `this.design()` method to get a reference to the form's design and then we check whether the first control in the design is an action pane. We continue by adding a new separate button group and the **About Dynamics 365 for Operations** command button. Now, every form in Dynamics 365 for Finance and Operations with an action pane will have one more button.

Storing the last form values

Dynamics 365 for Finance and Operations has a very useful feature that allows you to save the latest user choices per user per form, report, or any other object. This feature is implemented across a number of standard forms, reports, periodic jobs, and other objects which require user input. When developing a new functionality for Dynamics 365 for Finance and Operations, it is recommended that you keep it that way.

In this recipe, we will demonstrate how to save the latest user selections. In order to make it as simple as possible, we will use the existing filters on the **Bank statement** form, which can be opened by navigating to **Cash and bank management** | **Common** | **Bank accounts**, selecting any bank account, and then clicking on the **Account reconciliation** button in the **Action** pane. This form contains one filter control called **View**, which allows you to display bank statements based on their status. The default view of this form is **Unreconciled**. We will see how to use the below code to save user selections for future purposes.

How to do it...

Carry out the following steps in order to complete this recipe:

1. In the AOT, find the `BankAccountStatement` form and add the following code snippet to the bottom of its class declaration:

   ```
   AllNotReconciled showAllReconciled;
   #define.CurrentVersion(1)
   #localmacro.CurrentList
   showAllReconciled
   #endmacro
   ```

2. Add the following additional form methods:

   ```
       public void initParmDefault()
       {
        showAllReconciled = AllNotReconciled::NotReconciled;
       }

       public container pack()
       {
         return [#CurrentVersion, #CurrentList];
       }

       public boolean unpack(container _packedClass)
       {
         int version = RunBase::getVersion(_packedClass);

           switch (version)
           {
             case #CurrentVersion:
             [version, #CurrentList] = _packedClass;
             return true;
             default:
             return false;
           }
   ```

```
        return false;
    }

    public IdentifierName lastValueDesignName()
    {
      return element.args().menuItemName();
    }

    public IdentifierName lastValueElementName()
    {
      return this.name();
    }

    public UtilElementType lastValueType()
    {
      return UtilElementType::Form;
    }

    public UserId lastValueUserId()
    {
      return curUserId();
    }

    public DataAreaId lastValueDataAreaId()
    {
      return curext();
    }
```

3. Override the form's `run()` method and add the following lines of code right before its `super()` method:

```
xSysLastValue::getLast(this);
AllReconciled.selection(showAllReconciled);
```

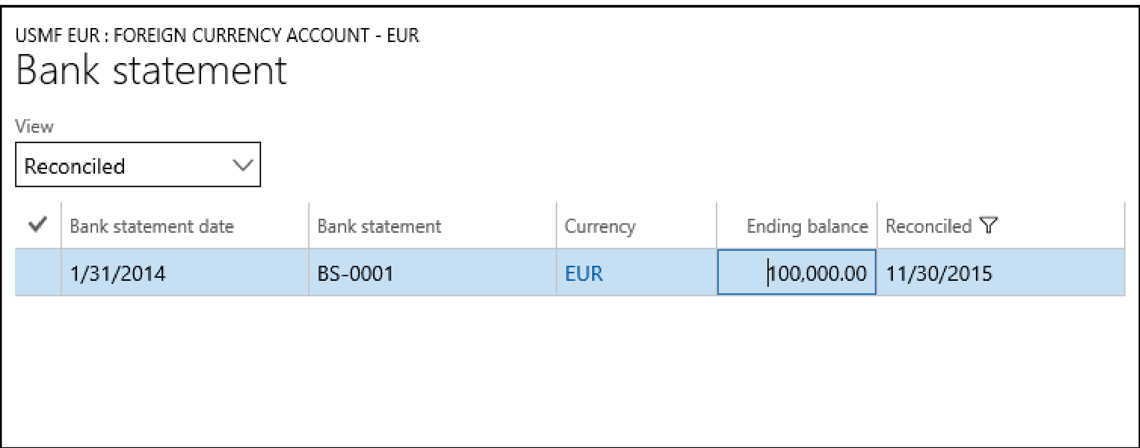
4. Override the form's `close()` method and add the following lines of code at the bottom of this method:

```
showAllReconciled = AllReconciled.selection();
xSysLastValue::saveLast(this);
```

5. Finally, delete the following line of code from the `init()` method of the `BankAccountStatement` data source:

```
allReconciled.selection(1);
```

6. Now, to test the form, navigate to **Cash and bank management** | **Common** | **Bank accounts**, select any bank account, click on **Account reconciliation**, change the filter's value, close the form, and then open it again. The latest selection should remain, as shown in the following screenshot:



How it works...

First, we define a variable that will store the value of the filter control. The `#CurrentList` macro is used to define a list of variables that we are going to save in the usage data. Currently, we have our single variable inside it.

The `#CurrentVersion` macro defines a version of the saved values. In other words, it says that the variables defined by the `#CurrentList` macro, which will be stored in the system usage data, can be addressed using the number `1`.

Normally, when implementing the last value saved for the first time for a particular object, `#CurrentVersion` is set to `1`. Later on, if you decide to add new values or change the existing ones, you have to change the value of `#CurrentVersion`, normally increasing it by one. This ensures that the system addresses the correct list of variables in the usage.

The `initParmDefault()` method specifies the default values if nothing is found in the usage data. Normally, this happens if we run a form for the first time, we change `#CurrentVersion`, or we clear the usage data. This method is called automatically by the `xSysLastValue` class.

The `pack()` and `unpack()` methods are responsible for formatting a storage container from the variables and extracting variables from a storage container, respectively. In our case, `pack()` returns a container consisting of two values: version number and statement status. These values will be sent to the system usage data storage after the form is closed. When the form is opened, the `xSysLastValue` class uses `unpack()` to extract values from the stored container. It checks whether the container version in the usage data matches the current version number defined by `#CurrentVersion`, and only then the values are considered correct and assigned to the form's variables.

The return values of `lastValueDesignName()`, `lastValueElementName()`, `lastValueType()`, `lastValueUserId()`, and `lastValueDataAreaId()` represent a unique combination that is used to identify the stored usage data. This ensures that different users can store the last values of different objects in different companies without overriding each other's values.

The `lastValueDesignName()` method is meant to return the name of the object's current design in cases where the object can have several designs. In this recipe, there is only one design, so instead of leaving it empty, we used it for a slightly different purpose. The method returns the name of the menu item used to open this form. In this case, separate usage datasets will be stored for each menu item that opens the same form.

The last two pieces of code need to be added to the form's `run()` and `close()` methods. In the `run()` method, `xSysLastValue::getLast(this)` retrieves the saved user values from the usage data and assigns them to the form's variables.

Finally, the code in the `close()` method is responsible for assigning user selections to the variables and saving them to the usage data by calling `xSysLastValue::saveLast(this)`.

Using a tree control

Frequent users will notice that some of the Dynamics 365 for Finance and Operations forms use tree controls instead of the commonly used grids. In some cases, this is extremely useful, especially when there are parent-child relationships among records. It is a much clearer way to show the whole hierarchy, as compared to a flat list. For example, product categories are organized as a hierarchy and give a much better overview when displayed in a tree layout.

This recipe will discuss the principles of how to build tree-based forms. As an example, we will use the **Budget model** form, which can be found by navigating to **Budgeting** | **Setup** | **Basic Budgeting** | **Budget models**. This form contains a list of budget models and their submodels and, although the data is organized using a parent-child structure, it is still displayed as a grid. In this recipe, in order to demonstrate the usage of the `Tree` control, we will replace the grid with a new `Tree` control.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Add a new project in your solution in Visual Studio. Add a new class called `BudgetModelTree` with the following code snippet:

```
public class BudgetModelTree
{
  FormTreeControl tree;
  BudgetModelId   modelId;
}

public void new(
  FormTreeControl _formTreeControl,
  BudgetModelId   _budgetModelId)
{
  tree    = _formTreeControl;
  modelId = _budgetModelId;
}

public static BudgetModelTree construct(
    FormTreeControl _formTreeControl,
    BudgetModelId   _budgetModelId = '')
{
   return new BudgetModelTree(
    _formTreeControl,
     _budgetModelId);
}

private TreeItemIdx createNode(
   TreeItemIdx   _parentIdx,
   BudgetModelId _modelId,
   RecId         _recId)
{
  TreeItemIdx itemIdx;
  BudgetModel model;
  BudgetModel submodel;
```

```
  model = BudgetModel::find(HeadingSub::Heading,
   _modelId);

  itemIdx = SysFormTreeControl::addTreeItem(
  tree,
  _modelId + ' : ' + model.Txt,
  _parentIdx,
  _recId,
   0,
   true);
   if (modelId == _modelId)
    {
      tree.select(itemIdx);
    }
    while select submodel
    where submodel.ModelId == _modelId &&
    submodel.Type    == HeadingSub::SubModel
    {
      this.createNode(
      itemIdx,
      submodel.SubModelId,
      submodel.RecId);
    }
    return itemIdx;
}

public void buildTree()
{
  BudgetModel model;
  BudgetModel submodel;
  TreeItemIdx itemIdx;

  tree.deleteAll();
  tree.lock();
  while select RecId, ModelId from model
  where model.Type == HeadingSub::Heading
  notExists join submodel
  where submodel.SubModelId == model.ModelId &&
  submodel.Type       == HeadingSub::SubModel
    {
      itemIdx = this.createNode(
      FormTreeAdd::Root,
      model.ModelId,
      model.RecId);
      SysFormTreeControl::expandTree(tree, itemIdx);
    }
    tree.unLock(true);
}
```

2. In the AOT, open the `BudgetModel` form's design, expand the `Body` group, then expand the `GridContainer` group and change the following property of the `BudgetModel` grid control:

Property	Value
Visible	No

3. Create a new `Tree` control right below the `BudgetModel` grid with these properties, as shown in the following table along with their values:

Property	Value
Name	Tree
Width	Column width
Height	Column height
Border	Single line
RowSelect	Yes
AutoDeclartion	Yes

4. Add the following line of code to the bottom of the form's class declaration:

```
BudgetModelTree modelTree;
```

5. Add the following lines of code at the bottom of the form's `init()` method:

```
modelTree = BudgetModelTree::construct(Tree);
modelTree.buildTree();
```

6. Override `selectionChanged()` on the `Tree` control with the following code snippet:

```
public void selectionChanged(
    FormTreeItem   _oldItem,
    FormTreeItem   _newItem,
    FormTreeSelect _how)
{
  BudgetModel   model;
  BudgetModelId modelId;

    super(_oldItem, _newItem, _how);
```

```
        if (_newItem.data())
        {
          select firstOnly model
          where model.RecId == _newItem.data();
            if (model.Type == HeadingSub::SubModel)
            {
              modelId = model.SubModelId;
              select firstOnly model
              where model.ModelId == modelId
              && model.Type    ==  HeadingSub::Heading;
            }
            BudgetModel_ds.findRecord(model);
            BudgetModel_ds.refresh();
        }

    }
```

7. Override the `delete()` method on the `BudgetModel` data source with the following code snippet:

```
public void delete()
{
  super();

    if (BudgetModel.RecId)
    {
      modelTree.buildTree();
    }
}
```

8. Add the following line of code at the bottom of the `write()` method on the `BudgetModel` data source:

```
modelTree.buildTree();
```

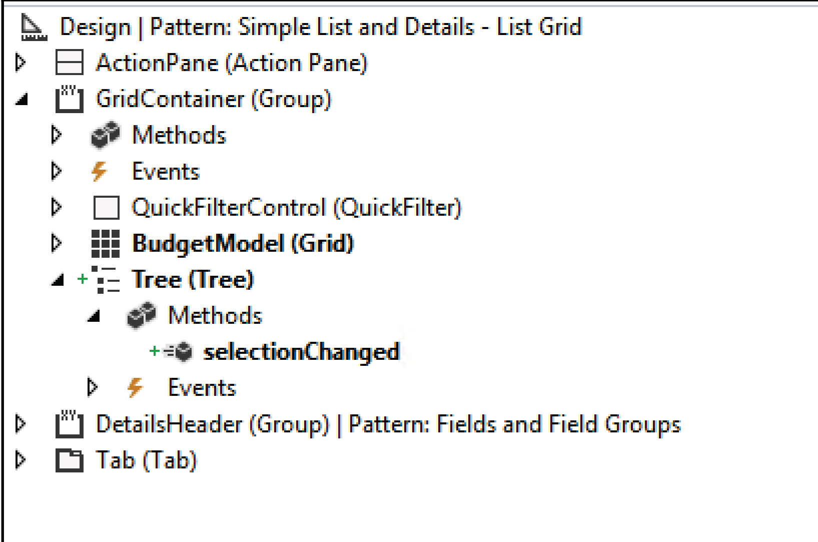
9. Override the `delete()` method on the `SubModel` data source with the following code snippet:

```
public void delete()
{
  super();

    if (SubModel.RecId)
    {
      modelTree.buildTree();
    }
}
```

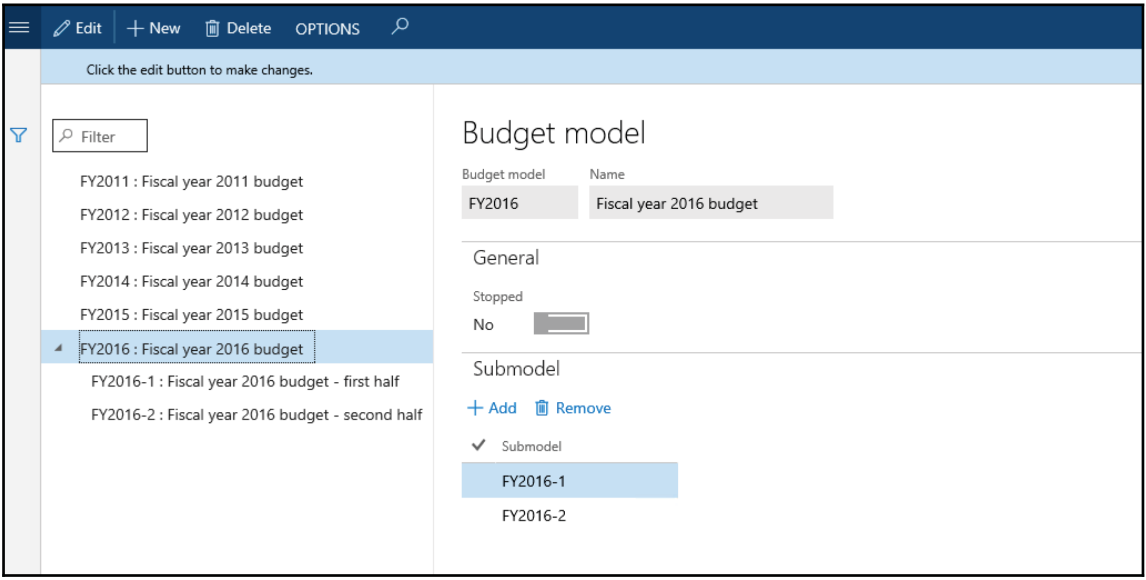
10. Override the `write()` method on the `SubModel` data source and add the following line of code at the bottom:

```
modelTree.buildTree();
```

11. Save all your code and build your solution.
12. In Visual Studio, the `BudgetModel` design should look like the following screenshot:



13. To test the `Tree` control, navigate to **Budgeting** | **Setup** | **Basic budgeting** | **Budget models**. Notice how the budget models are presented as a hierarchy, as shown here:



How it works...

This recipe contains a lot of code, so we create a class to hold most of it. This allows you to reuse the code and keep the form less cluttered.

The new class contains a few common methods, such as `new()` and `construct()`, for initializing the class, and two methods which actually generate the tree.

The first method is `createNode()` and is used to create a single budget model node with its children, if any. It is a recursive method, and it calls itself to generate the children of the current node. It accepts a parent node and a budget model as arguments. In this method, we create the node by calling the `addTreeItem()` method of the `SysFormTreeControl` class. The rest of the code loops through all the submodels and creates subnodes (if there are any) for each of them.

Second, we create `buildTree()`, where the whole tree is created. Before we actually start building it, we delete all the existing nodes (if any) in the tree and then lock the `Tree` control to make sure that the user cannot modify it while it's being built. Then, we add nodes by looping through all the parent budget models and calling the previously mentioned `createNode()`. We call the `expandTree()` method of the `SysFormTreeControl` class in order to display every parent budget model that was initially expanded. Once the hierarchy is ready, we unlock the `Tree` control.

Next, we modify the **BudgetModel** form by hiding the existing grid section and adding a new `tree` control. Tree nodes are always generated from the code and the previously mentioned class will do exactly that. On the form, we declare and initialize the `modelTree` object and build the tree in the form's `init()` method.

In order to ensure that the currently selected tree node is displayed on the form on the right-hand side, we override the `tree` control's `selectionChanged()` event, which is triggered every time a tree node is selected. Here, we locate a corresponding record and place a cursor on that record.

The rest of the code on the form is to ensure that the tree is rebuilt whenever the data is modified.

See also

- The *Preloading images* recipe in `Chapter 3`, *Working with Data in Forms*
- The *Building a tree lookup* recipe in `Chapter 4`, *Building Lookups*

Adding the View details link

Dynamics 365 for Finance and Operations has a very useful feature that allows the user to open the main record form with just a few mouse clicks on the current form. The feature is called **View details** and is available in the right-click context menu on some controls. It is based on table relationships and is available for those controls whose data fields have foreign key relationships with other tables.

Because of the data structure's integrity, the **View details** feature works most of the time. However, when it comes to complex table relations, it does not work correctly or does not work at all. Another example of when this feature does not work automatically is when the `display` or `edit` methods are used on a form. In these and many other cases, the **View details** feature has to be implemented manually.

In this recipe, to demonstrate how it works, we will modify the **General journal** form in the **General ledger** module and add the **View details** feature to the `Description` control, allowing users to jump from the right-click context menu to the **Journal names** form.

How to do it...

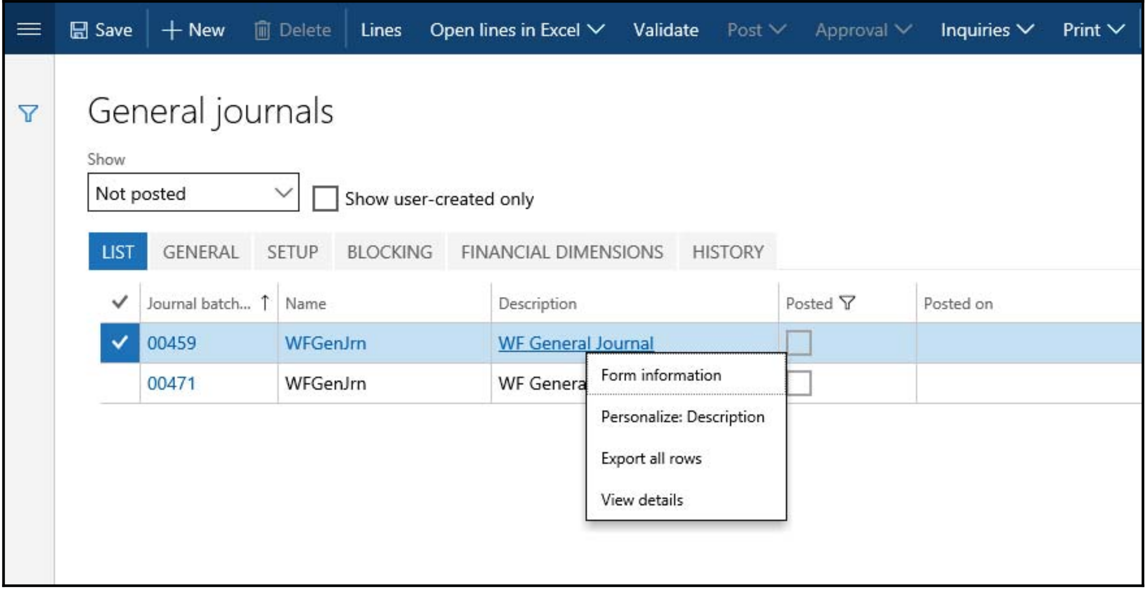
Carry out the following steps in order to complete this recipe:

1. Add the `LedgerJournalTable` form to your project, expand its data sources, and override `jumpRef()` of the `Name` field on the `LedgerJournalTable` data source with the following code snippet:

```
public void jumpRef()
{
  LedgerJournalName   name;
  Args                args;
  MenuFunction        mf;

  name = LedgerJournalName::find
   (LedgerJournalTable.JournalName);

  if (!name)
  {
    return;
  }

  args = new Args();
  args.caller(element);
  args.record(name);

  mf = new MenuFunction
  (menuitemDisplayStr(LedgerJournalSetup),
  MenuItemType::Display);
  mf.run(args);
}
```

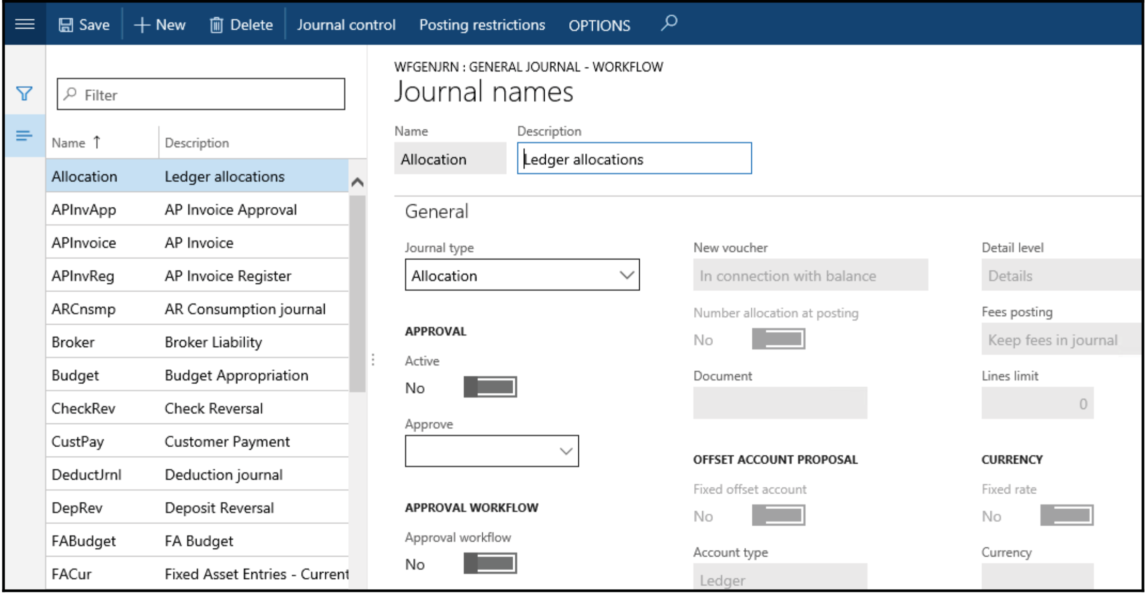
2. Save all your code and build your solution.

3. Navigate to **General ledger** | **Journals Entries** | **General journal**, select any of the existing records, and right-click on the **Description** column. Notice that the **View details** option, which will open the **Journal names** form, is available now, as shown here:



You may need to refresh your Dynamics 365 for Operations page to reflect your changes in the frontend.

4. When you click on **View details**, the below form should open:



How it works...

Normally, the **View details** feature is controlled by the relationships between the underlying tables. If there are no relationships or the form control is not bound to a table field, then this option is not available. However, we can force this option to appear by overriding the control's `jumpRef()` method.

In this method, we add code that opens the relevant form. This can be done by declaring, instantiating, and running a `FormRun` object, but an easier way to do this is to simply run the relevant menu item from the code. In this recipe, the code in `jumpRef()` does exactly that.

In the code, first we check whether a valid journal name record is found. If so, we run the `LedgerJournalSetup` menu item with an `Args` object that holds the journal name record and the current `form` object information as a caller. The rest is done automatically by the system, that is, the **Journal names** form is opened with the currently selected journal name.

Selecting a form pattern

In the latest version of Dynamics 365 for Finance and Operations, form patterns are now an integrated part of the form development experience. These patterns provide form structure based on a particular style (including required and optional controls), and also provide many default control properties. In addition to top-level form patterns, Dynamics 365 for Operations has also introduced subpatterns which can be applied to container controls, and that provide guidance and consistency for subcontent on a form, such as, on a **Fast Tab**.

Form patterns have made form development easier in Dynamics 365 for Finance and Operations by providing a guided experience for applying patterns to forms to guarantee that they are correct and consistent. Patterns help validate form and control structures, and also the use of controls in some places. Patterns also help guarantee that each new form that a user encounters is immediately recognizable in appearance and function. Form patterns can provide many default control properties, and these also contribute to a more guided development experience. Because patterns provide many default layout properties, they help guarantee that forms have a responsive layout. Finally, patterns also help guarantee better compatibility with upgrades.

Many of the existing form styles and templates from AX 2012 continue to be supported in the current version Dynamics 365 for Finance and Operations. However, legacy form styles and templates that aren't supported have a migration path to a Dynamics 365 for Finance and Operations pattern. Because the foundational elements of Dynamics 365 for Finance and Operations are built based on those legacy form styles and patterns, the transition from AX 2012 to the current version of Dynamics 365 for Finance and Operations is as easy as possible.

The selection of a form pattern is an important step in the process of migrating a form. A pattern that is a good fit for the target form reduces the amount of migration work that is required. Therefore, some investigation is required to select the best form pattern for the form that you're migrating.

How to do it

Applying a pattern is a straightforward process that can modify properties on multiple containers and controls on a form. Here is the standard flow for applying patterns:

1. Identify a target form and add it to your project. Then, in Visual Studio, open **Application Explorer**, and find the form. Right-click the form, and then select **Add to project**. When you open the form in the designer, it should have the **Pattern: <unselected>** designation on the design node.

2. Decide which pattern to apply. You can refer to the exported details file in the last recipe.
3. Now we need to apply the pattern. Right-click the **Design node** of the target form, select **Apply pattern**, and then click the **Pattern to apply**.
4. As a last step, we may need to handle a few errors. Information about the pattern appears on the **Pattern** tab. To learn about the pattern structure, click **control names** on the **Pattern** tab to navigate the pattern structure.
5. Double-click an error to go to the control that the error was reported for, if the control exists.
6. If the control already exists on the form but is in a different place, move the control to the correct place, as indicated by the pattern.
7. If the control doesn't exist, create the control.

Full list of form patterns

In the current version of Dynamics 365 for Finance and Operations, there are a total of five form patterns that we use the most:

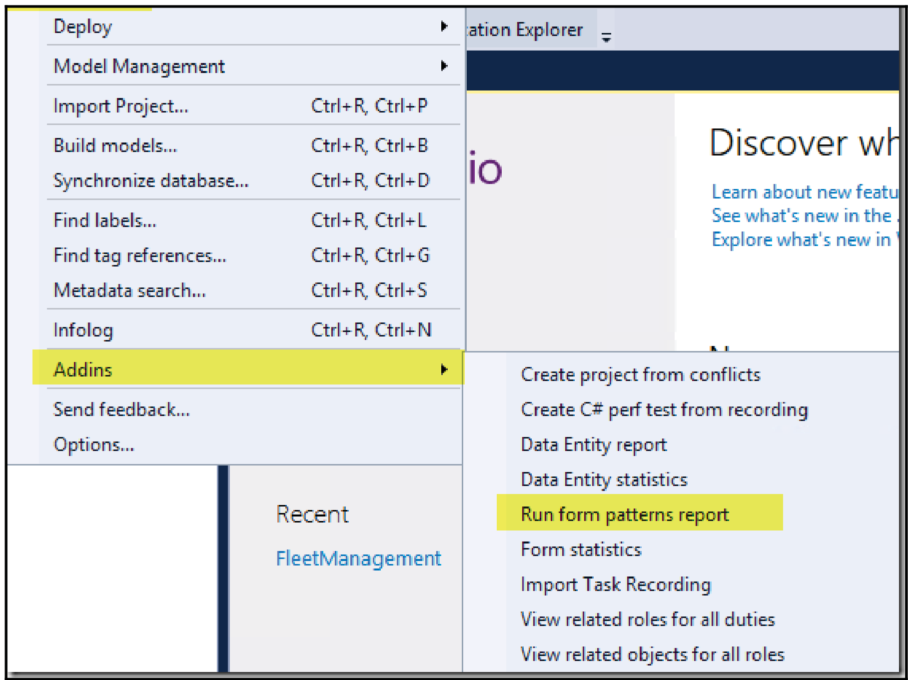
- Details Master
- Form Part - Fact Boxes
- Simple List
- Table of Contents
- Operational workspaces

For a full list of the forms that are currently using a particular form pattern, generate the **Form Patterns** report from within Microsoft Visual Studio. On the Dynamics 365 menu, expand the **Add-ins** option, and click **Run form** patterns report. A background process generates the report. After several seconds, a message box appears in Visual Studio to indicate that the report has been generated and inform you about the location of the **Form Patterns** report file. You can filter this file by pattern to find forms that use a particular pattern.

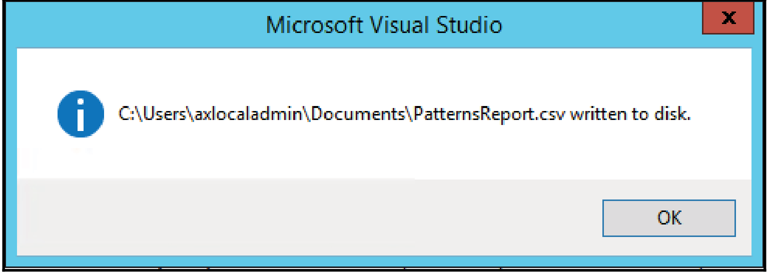
How to do it...

We're going to look at how to run the form patterns report. This report is generated through Visual Studio and gives us information about all of the forms in the system with the corresponding form patterns:

1. Open VS as admin and go to the **Dynamics 365** menu.
2. Select **Add ins** | **Run form patterns report**:



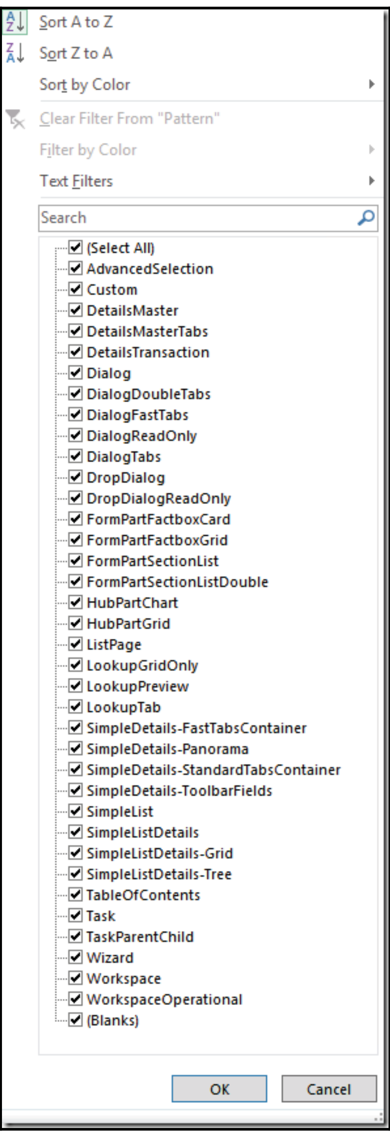
3. This will generate a CSV file that we can open in Excel.



4. Open the Excel file and you'll notice that, we have columns that tell us which model the form is in, the system name for the form, as well as the form style and the form pattern:

Model	Form	Style
UnitOfMeasure	UnitOfMeasureLookup	Lookup
UnitOfMeasure	UnitOfMeasure	SimpleListDetails
Foundation	PdsCustRebateGroup	SimpleList
Tutorial	Tutorial_ControlForm	Auto
Foundation	VendInvoiceInfoListPagePreviewPane	FormPart
Foundation	BudgetTransactionInquiry	SimpleList
Foundation	PdsCostBasis	SimpleList
UnitOfMeasure	UnitOfMeasureTranslation	SimpleList
Foundation	JmgFlexBalance	SimpleList
TestEssentials	HierarchicalGridTest	Auto
Foundation	FactureCorrectedAmounts_RU	Auto
Foundation	RDeferralsProfileTrans	SimpleList
Foundation	BudgetTransactionHeaderWorkflowDropDialog	DropDialog
Foundation	FactureEditLines_RU	Auto
Foundation	SalesLineDeliveryDetails	Dialog
Foundation	JmgFeedbackWizard	Dialog
Foundation	PdsCustSellableDays	SimpleList
Foundation	PdsFreightGroup	SimpleList
TestEssentials	TestFormLauncher	Auto
Foundation	FactureJourLookup_RU	Lookup
Foundation	VendInvoiceInfoSubTableLookup	Lookup

5. In this CSV file, we will also get details about the controls and the coverage percentage. Filter helps to determine which system form has which pattern applied, and also which forms do not have any pattern applied. The following are all form patterns in Dynamics 365 for Operations:



Apply your filter as per your requirements and see the results.

Creating a new form

In Dynamics 365 for Finance and Operations, form creations are slightly easier than in AX2012 and earlier versions. Here, we have more tools to create any specific form using design templates. Every form plays an important role where we need to interact with the user to view, insert, update, or delete any record(s).

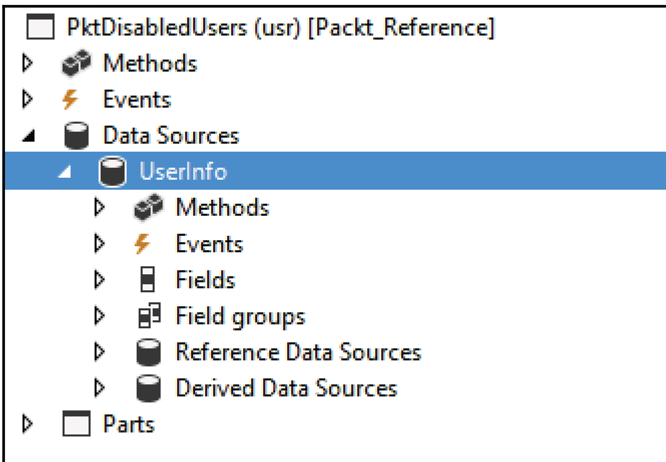
In this recipe, we will create a simple form using a template and add this form to one of the menus so that users can access it from the front end.

Getting ready

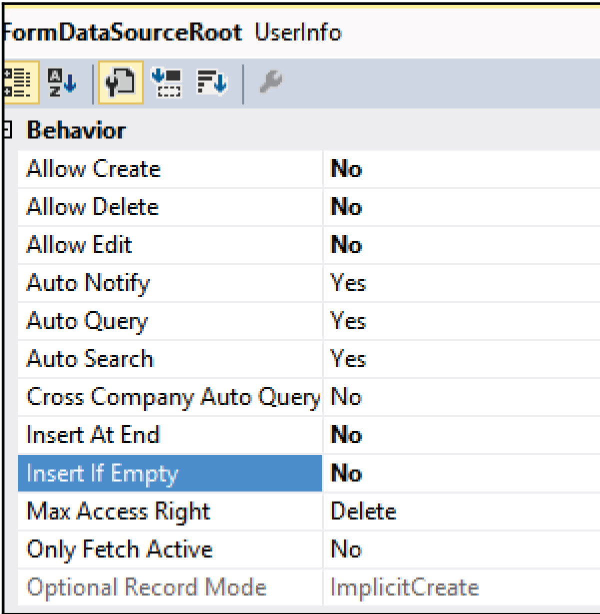
Let's think about a scenario where the admin needs to check all existing users in the system. Although we have one standard form for this, we cannot give access to everyone because this form also has many other options to perform on this form, while our requirement is to just see read only data. We will use this form in further recipes to justify the name of this form. Here, it will show all enabled and disabled users.

How to do it...

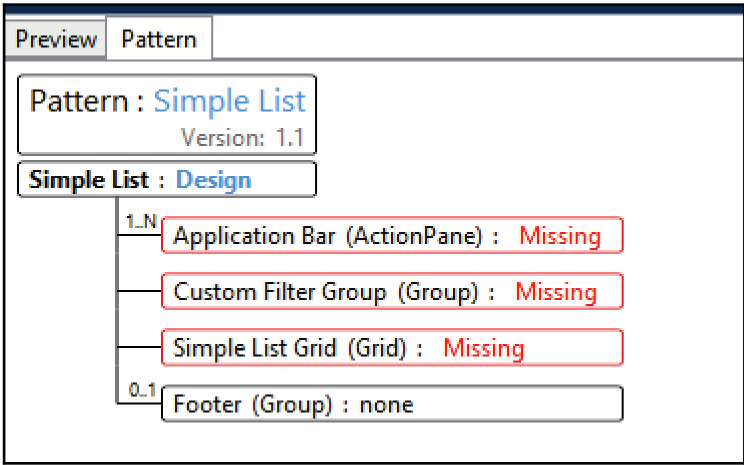
1. Add a new form in your project name it `PktDisabledUsers`.
2. Expand the `Data Sources` node and add the `UserInfo` table in it:



3. Set the following properties on **Data source**:

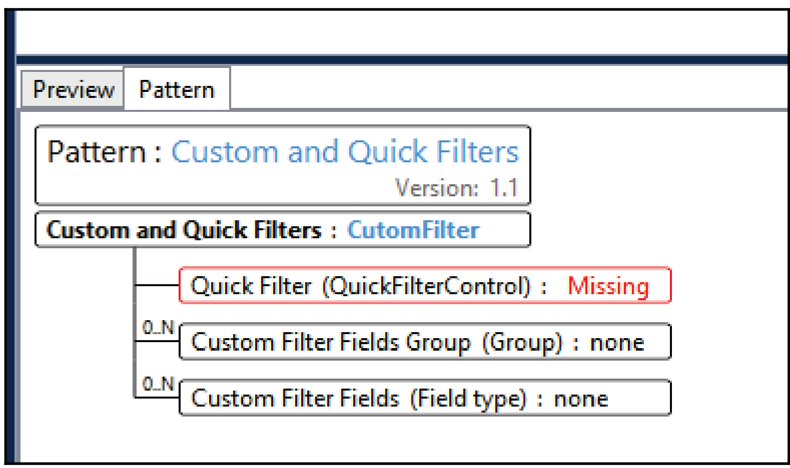


4. Go to **Design**, right click on **Apply Pattern**, and select **Simple List**. This will create the respective pattern for you:

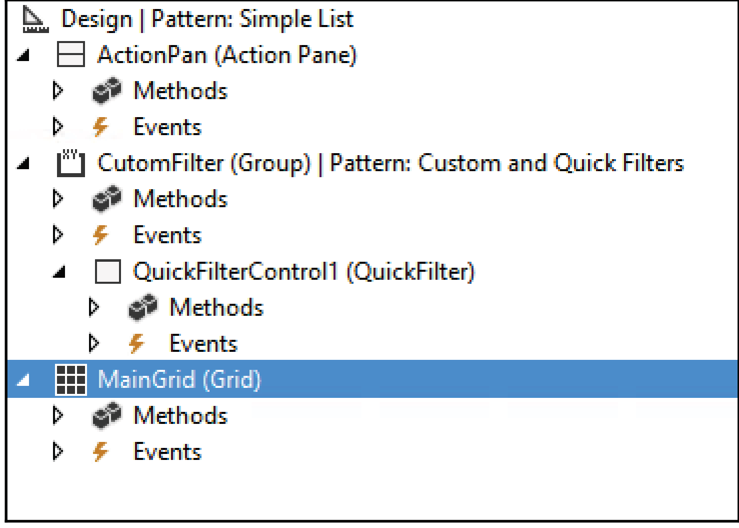


5. You have to add all those elements in design the section in the same order.

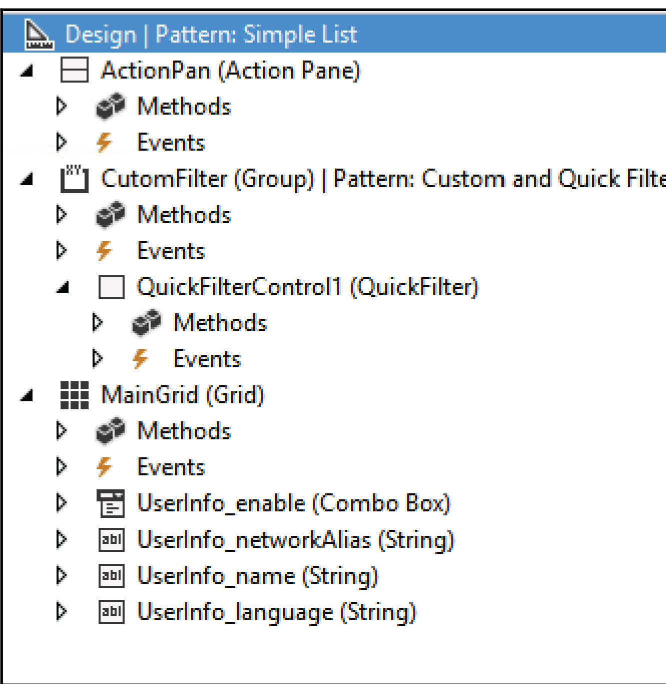
6. Add all the missing objects such as `ActionPane`, `Group`, and a `Grid`.
7. Now, you have to select a pattern for **CustomFilter** group; select **Custom** and **Quick Filters**.
8. You will see another pattern available for this; add all missing objects to this group:



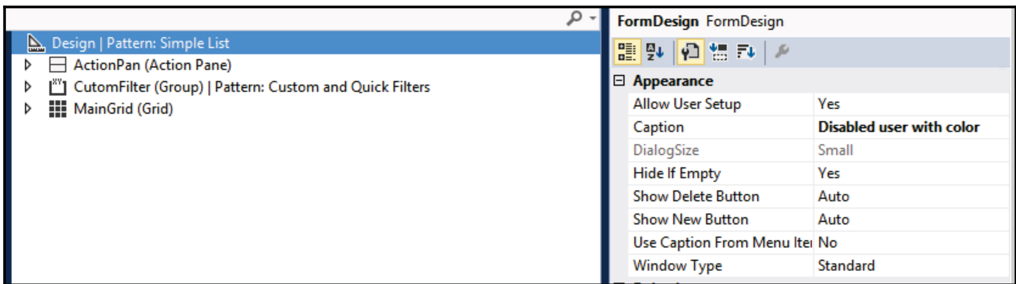
9. Your form design should look like this:



10. Now, let's add some fields in your grid to show actual data. Add fields from **UserInfo Data source**:

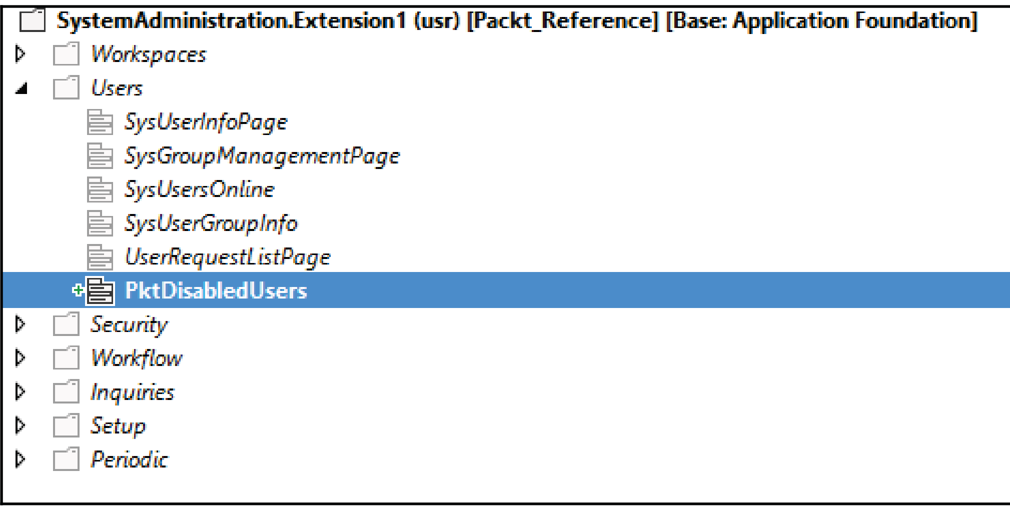


11. Give a caption to your design, such as `Disabled users with color`. We will use this form in further recipes to justify its name:

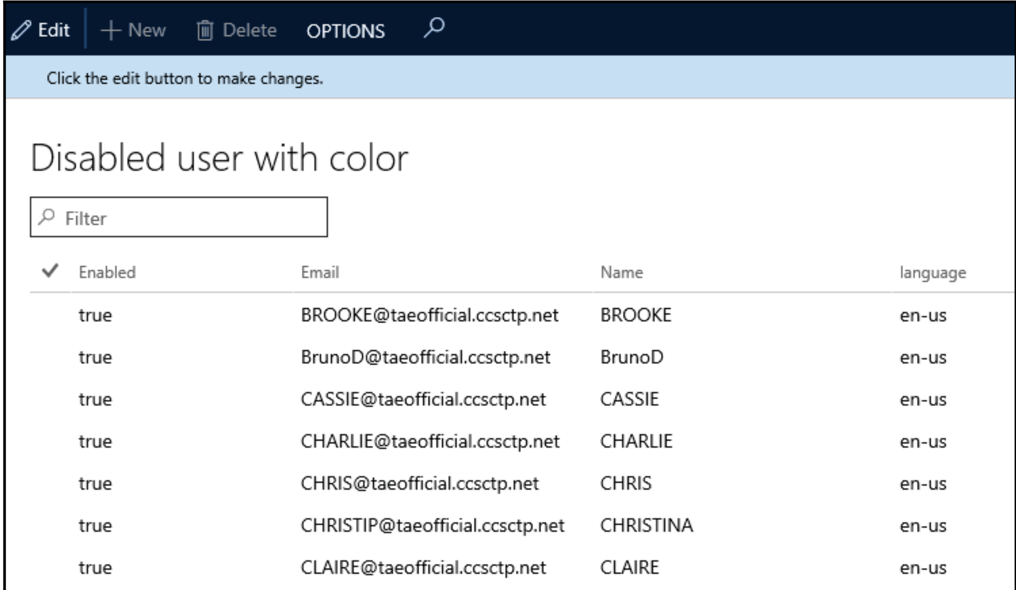


12. Now, to add this form to the front end, create a `Display Menu Item` for this form.

13. Create an extension of `SystemAdminstration` menu and add this new display menu item under **Users,** as in the following screenshot:



14. To test this form, save all your changes and build the solution. Now, go to **System administrator| Users |Disabled users with color**. Your form should look like the following screenshot:



How it works...

Creating a new form in Dynamics 365 for Finance and Operations is very systematic and easy. All you need to do is identify the purpose of the form and choose a relevant pattern. Once you apply a pattern to your form, it will show the required object details with a sequence, as shown in step 4 in the above recipe. All you need to add now are the respective objects in your design with the given sequence.

3

Working with Data in Forms

In this chapter, we will cover the following recipes:

- Using a number sequence handler
- Creating a custom filter control
- Creating a custom instant search filter
- Building a selected/available list
- Creating a wizard
- Processing multiple records
- Coloring records
- Adding an image to records

Introduction

This chapter basically supplements the previous one and explains data organization in forms in the new Dynamics 365 for Finance and Operations. It shows how to add custom filters to forms to allow users to filter data and create record lists for quick data manipulation.

This chapter also discusses how the displaying of data can be enhanced by adding icons to record lists and trees, and how normal images can be stored along with the data by reusing the existing Dynamics 365 for Finance and Operations application objects.

A couple of recipes will show you how to create wizards in the new Dynamics 365 for Finance and Operations to guide users through complex tasks. This chapter will also show several approaches to capturing user-selected records on forms for further processing, and ways to distinguish specific records by coloring them.

Using a number sequence handler

As already discussed in the *Creating a new number sequence* recipe in `Chapter 1`, *Processing Data*, number sequences are widely used throughout the system as a part of the standard application. Dynamics 365 for Finance and Operations also provides a special number sequence handler class to be used in forms. It is called `NumberSeqFormHandler` and its purpose is to simplify the usage of record numbering on the user interface. Some of the standard Dynamics 365 for Finance and Operations forms, such as **Customers** or **Vendors**, already have this feature implemented.

This recipe shows you how to use the number sequence handler class. Although in this demonstration we will use an existing form, the same approach will be applied when creating brand new forms.

For demonstration purposes, we will use the existing **Customer groups** form located in **Accounts receivable** | **Setup** | **Customers** and change the **Customer group** field from manual to automatic numbering. We will use the number sequence created earlier, in the *Creating a new number sequence* recipe in `Chapter 1`, *Processing Data*.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Create a new project, `UsingNumberSeqhandler`, create a new extension class `CustGroup_Extension` for the `CustGroup` form, and add the following code snippet to its class declaration:

```
public NumberSeqFormHandler numberSeqFormHandler;
```

2. Also, create a new method called `numberSeqFormHandler()` in the same class:

```
public NumberSeqFormHandler numberSeqFormHandler()
{
  if (!numberSeqFormHandler)
  {
    numberSeqFormHandler = NumberSeqFormHandler::newForm(
    CustParameters::numRefCustGroupId().NumberSequenceId,
    this,this.CustGroup_ds,fieldNum(CustGroup,CustGroup));
  }
  return numberSeqFormHandler;
}
```

3. To override the `CustGroup` data source's `create()` method, copy the `OnCreating` and `OnCreated` events from the data source and paste them in the class `CustGroup_Extension` with the following code snippet:

```
[FormDataSourceEventHandler(formDataSourceStr(CustGroup,
CustGroup), FormDataSourceEventType::Creating)]
public void CustGroup_OnCreating(FormDataSource sender,
FormDataSourceEventArgs e)
{
   this.numberSeqFormHandler().formMethodDataSourceCreatePre();
}

[FormDataSourceEventHandler(formDataSourceStr(CustGroup,
CustGroup), FormDataSourceEventType::Created)]
public void CustGroup_OnCreated(FormDataSource sender,
FormDataSourceEventArgs e)
{
   this.numberSeqFormHandler().formMethodDataSourceCreate();
}
```

4. Then, to override its `delete()` method, subscribe to the `OnDeleting` event of the `CustGroup` data source and paste it in the `extension` class with the following code snippet:

```
[FormDataSourceEventHandler(formDataSourceStr(CustGroup,
CustGroup), FormDataSourceEventType::Deleting)]
public void CustGroup_OnDeleting(FormDataSource sender,
FormDataSourceEventArgs e)
{
   this.numberSeqFormHandler().formMethodDataSourceDelete();
}
```

5. Then, to override the data source's `write()` method, subscribe to the `OnWritten` event with the following code snippet:

```
 [FormDataSourceEventHandler(formDataSourceStr(CustGroup,
 CustGroup), FormDataSourceEventType::Written)]
 public void CustGroup_OnWritten(FormDataSource sender,
 FormDataSourceEventArgs e)
{
   this.numberSeqFormHandler().formMethodDataSourceWrite();
}
```

6. Similarly, to override its `validateWrite()` method, subscribe to the `OnValidatedWrite` event of the `CustGroup` data source with the following code snippet:

```
[FormDataSourceEventHandler(formDataSourceStr(CustGroup,
 CustGroup), FormDataSourceEventType::ValidatedWrite)]
public void CustGroup_OnValidatedWrite(FormDataSource sender,
 FormDataSourceEventArgs e)
{
  boolean ret = true;
  ret =
  this.numberSeqFormHandler().formMethodDataSourceValidateWrite
  (ret);

}
```

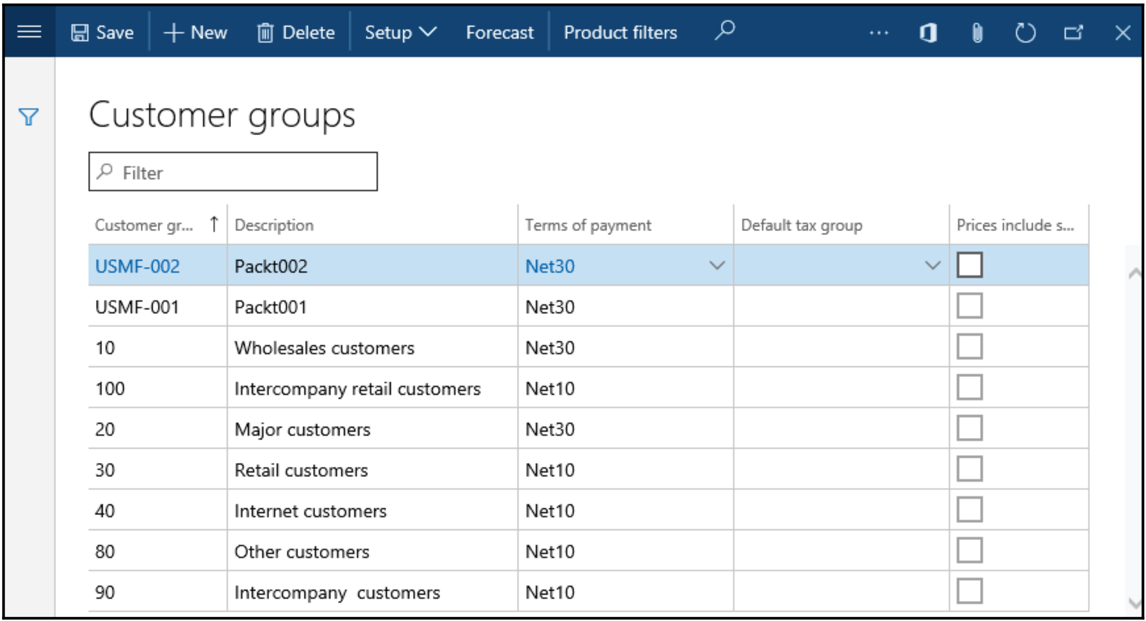
7. For the same data source, to override its `linkActive()` method, subscribe to `OnLinkActive` with the following code snippet:

```
[FormDataSourceEventHandler(formDataSourceStr(CustGroup,
 CustGroup), FormDataSourceEventType::PostLinkActive)]
 public void CustGroup_OnLinkActive(FormDataSource sender,
  FormDataSourceEventArgs e)
{
   this.numberSeqFormHandler()
   .formMethodDataSourceLinkActive();
}
```

8. Finally, to override the form's `close()` method, subscribe to the `OnClosing` event of the form with the following code snippet:

```
[FormEventHandler(formstr(CustGroup), FormEventType::Closing)]
 public void CustGroup_OnClosing(xFormRun formRun,
 FormEventArgs e)
{
  if (numberSeqFormHandler)
 {
     numberSeqFormHandler.formMethodClose();
 }
}
```

9. In order to test the numbering, navigate to **Accounts receivable** | **Setup** | **Customers** | **Customer groups** and try to create several new records--the **Customer group** value will be generated automatically:



How it works...

First, we declare an object of type `NumberSeqFormHandler` in the form's class declaration. Then, we create a new corresponding form method called `numberSeqFormHandler()`, which instantiates the object if it has not been instantiated yet and returns it. This method allows us to hold the handler creation code in one place and reuse it many times within the form.

In this method, we use the `newForm()` constructor of the `NumberSeqFormHandler` class to create the `numberSeqFormHandler` object. It accepts the following arguments:

The number sequence code, which was created in the *Creating a new number sequence* recipe in `Chapter 1`, *Processing Data*, ensures the proper format of the customer group numbering. Here, we call the `numRefCustGroupId()` helper method from the `CustParameters` table to find which number sequence code will be used when creating new customer group records:

- The `FormRun` object, which represents the form itself
- The form data source, where we need to apply the number sequence handler
- The field number of the field into which the number sequence will be populated

Finally, we add the various `NumberSeqFormHandler` methods to the corresponding events methods on the form's data source to ensure proper handling of the numbering when various events are triggered.

You may need to refresh your browser to see your changes.

See also

The *Creating a new number sequence* recipe in `Chapter 1`, *Processing Data*

Creating a custom filter control

Filtering forms in Dynamics 365 for Finance and Operations is implemented in a variety of ways. As part of the standard application, Dynamics 365 for Finance and Operations provides various filtering options, such as **Filter by Selection**, **Filter by Grid**, or **Advanced Filter/Sort** located in the toolbar, which allow you to modify the underlying query of the currently displayed form. In addition to the standard filters, the Dynamics 365 for Finance and Operations list pages normally allow quick filtering on most commonly used fields. Besides that, some of the existing forms have even more advanced filtering options, which allow users to quickly define complex search criteria.

Although the latter option needs additional programming, it is more user-friendly than standard filtering and is a very common request in most of the Dynamics 365 for Finance and Operations implementations.

In this recipe, we will learn how to add custom filters to a form. We will use the `Main accounts` form as a basis and add a few custom filters, which will allow users to search for accounts based on their name and type.

How to do it...

Carry out the following steps in order to complete this recipe:

1. In the AOT, locate the `MainAccount` form and select the option to `create extension` of it in a new project. Open the form and select the design under the `NavigationList` control and add a new group control with the following properties:

Property	Value
Name	Filter
AutoDeclaration	Yes

2. Move this group below `TreeFilter` and add a new `String` control with the following properties:

Property	Value
Name	FilterName
AutoDeclaration	Yes
ExtendedDataType	AccountName

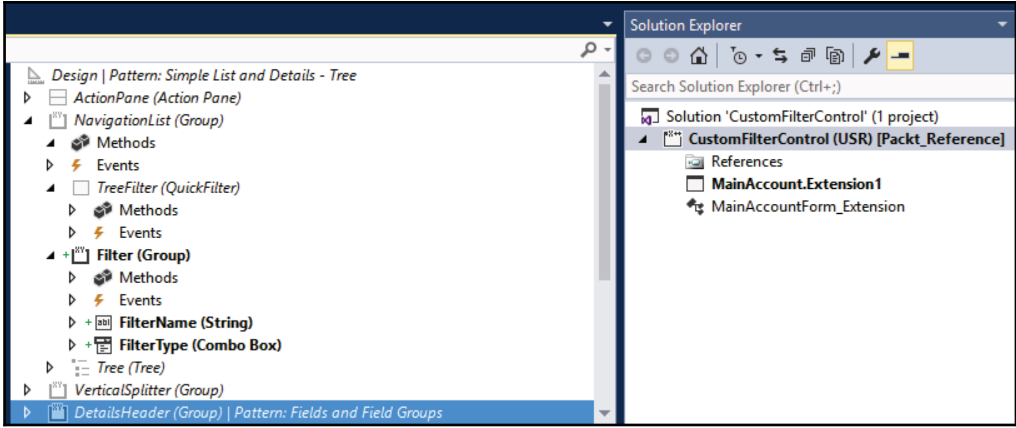
3. Add a new `ComboBox` control to the same group with the following properties:

Property	Value
Name	FilterType
AutoDeclaration	Yes
EnumType	DimensionLedgerAccountType
Selection	10

4. Create a new extension class, `MainAccountForm_Extension` , subscribe an `OnModified` event of the newly created controls, and add the following code snippet in the `extension` class:

```
   [ExtensionOf(formstr(MainAccount))]
   final class MainAccountForm_Extension
{
   [FormControlEventHandler(formControlStr(MainAccount,
   FilterName), FormControlEventType::Modified)]
   public void FilterName_OnModified(FormControl sender,
   FormControlEventArgs e)
{
   FormDataSource  mainAccount_ds =
   sender.formRun().dataSource(formdatasourcestr(MainAccount,
   MainAccount));
   mainAccount_ds.executeQuery();
}

   [FormControlEventHandler(formControlStr(MainAccount,
   FilterType), FormControlEventType::Modified)]
   public void FilterType_OnModified(FormControl sender,
   FormControlEventArgs e)
{
   FormDataSource  mainAccount_ds =
   sender.formRun().dataSource(formdatasourcestr(MainAccount,
   MainAccount));
   mainAccount_ds.executeQuery();
}
}
```

5. After all modifications, in the AOT, the `MainAccount.Extension1` form will look similar to the following screenshot:

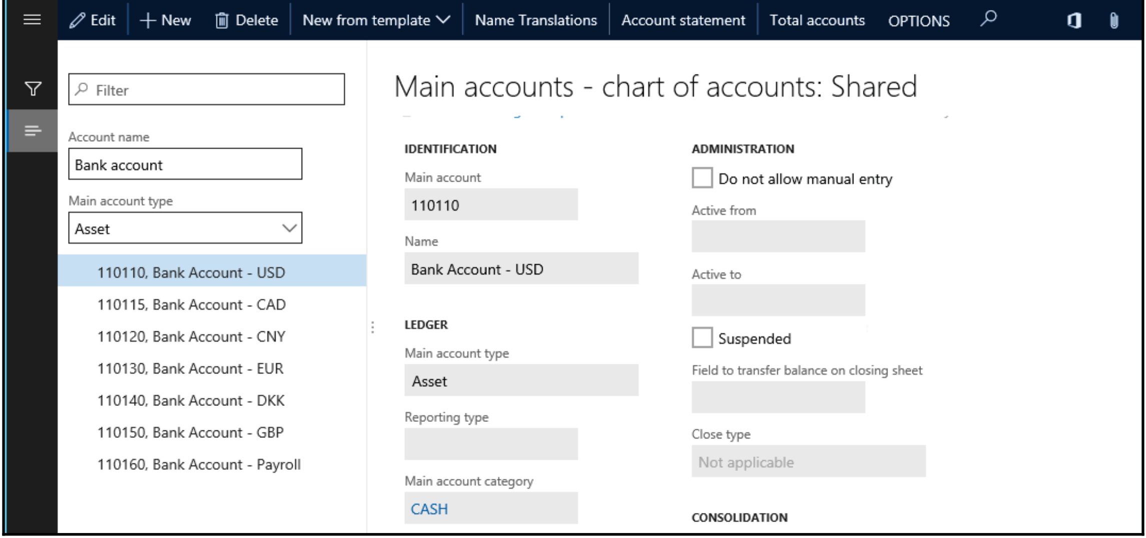


6. In the same extension class, subscribe to the `OnQueryExecuting` and `OnQueryExecuted` events to override the `executeQuery()` method of the `MainAccount` data source, and then add the following code snippet in the extension class:

```
[FormDataSourceEventHandler(formDataSourceStr(MainAccount,
MainAccount), FormDataSourceEventType::QueryExecuting)]
public void MainAccount_OnQueryExecuting(FormDataSource
sender, FormDataSourceEventArgs e)
 {
    QueryBuildRange qbrName;
    QueryBuildRange qbrType;
    QueryBuildDataSource qbds =
    sender.query().dataSourceTable(tableNum(MainAccount));

    MainAccount::updateBalances();

    qbrName = SysQuery::findOrCreateRange(
    qbds,fieldNum(MainAccount,Name));

    qbrType = SysQuery::findOrCreateRange(
    qbds,fieldNum(MainAccount,Type));

     str filterText =
     this.design().controlName("FilterName").valueStr();
     if (filterText)
     {
        qbrName.value(SysQuery::valueLike(
        filterText));
     }
     else
```

```
        {
           qbrName.value(SysQuery::valueUnlimited());
        }
        if (FilterType.selection() ==
             DimensionLedgerAccountType::Blank)
        {
           qbrType.value(SysQuery::valueUnlimited());
        }
        else
        {
           qbrType.value(queryValue(FilterType.selection()));
         }

 }

[FormDataSourceEventHandler(formDataSourceStr(MainAccount,
MainAccount), FormDataSourceEventType::QueryExecuted)]  public
void MainAccount_OnQueryExecuted(FormDataSource
sender, FormDataSourceEventArgs e)
{
   this.createTree();
}
```

- In order to test the filter, navigate to **General ledger** | **Common** | **Main accounts** and change the values in the newly created filters--the account list will change, reflecting the selected criteria:



How it works...

We start by adding an empty `Filter` group control to make sure all our controls are placed from the left to the right in one line.

Next, we add two controls that represent the **Account name** and **Main account type** filters and enable them to be automatically declared for later usage in the code. We also subscribe their `OnModified()` event methods to ensure that the `MainAccount` data source's query is re-executed whenever the controls' values change.

Finally, we subscribe to `OnQueryExecuting` and `OnQueryExecuted` events on the `MainAccount` data source to override the function of the `executeQuery()` method. The code uses two event handlers, to apply the ranges before `super()` of `executeQuery()` and `createTree()` after the `super().OnQueryExecuting` event make sure the query is modified before fetching the data.

Here, we declare and create two new `QueryBuildRange` objects, which represent the ranges on the query. We use the `findOrCreateRange()` method of the `SysQuery` application class to get the `range` object. This method is very useful and important, as it allows you to reuse previously created ranges.

Next, we set the ranges' values. If the filter controls are blank, we use the `valueUnlimited()` method of the `SysQuery` application class to clear the ranges. If the user types some text into the filter controls, we pass those values to the query ranges. The global `queryValue()` function--which is actually a shortcut to `SysQuery::value()`- ensures that only safe characters are passed to the range. The `SysQuery::valueLike()` method adds the `*` character around the account name value to make sure that the search is done based on partial text.

Note that the `SysQuery` helper class is very useful when working with queries, as it does all kinds of input data conversions to make sure they can be safely used. Here is a brief summary of some of the `SysQuery` methods:

- `valueUnlimited()`: This method returns a string representing an unlimited query range value, that is, no range at all.
- `value()`: This method converts an argument into a safe string. The global `queryValue()` method is a shortcut for this.
- `valueNot()`: This method converts an argument into a safe string and adds an inversion sign in front of it.

See also

- The *Building a query object* recipe in `Chapter 1`, *Processing Data.*

Creating a custom instant search filter

The standard form filters and the majority of customized form filters in Dynamics 365 for Finance and Operations are only applied once the user presses a button or a key. This is acceptable in most cases, especially if multiple criteria are used. However, when the result retrieval speed and usage simplicity has priority over system performance, it is possible to set up the search so that the record list is updated instantly when the user starts typing.

In this recipe, to demonstrate the instant search, we will modify the **Vendor group** form. We will add a custom **Name** filter, which will update the group list automatically when the user starts typing. We will need to overlay the **Vendor group** form, as the methods that we will be using for instant search don't yet have an event listener provided by Microsoft.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Create a new project in Visual Studio, open the `VendGroup` form, and add a new group control `Filter` below `QuickFilterControl` to the already existing `CustomFilterGroup` group control.

Property	**Value**
Name	Filter
AutoDeclaration	Yes

2. Then, add a `String` control to the new group `Filter` with the following properties:

Property	**Value**
Name	FilterName
AutoDeclaration	Yes
ExtendedDataType	Name

3. Override the control's `textChange()` method with the following code snippet:

```
public void textChange()
{
    super();

    VendGroup_ds.executeQuery();
}
```

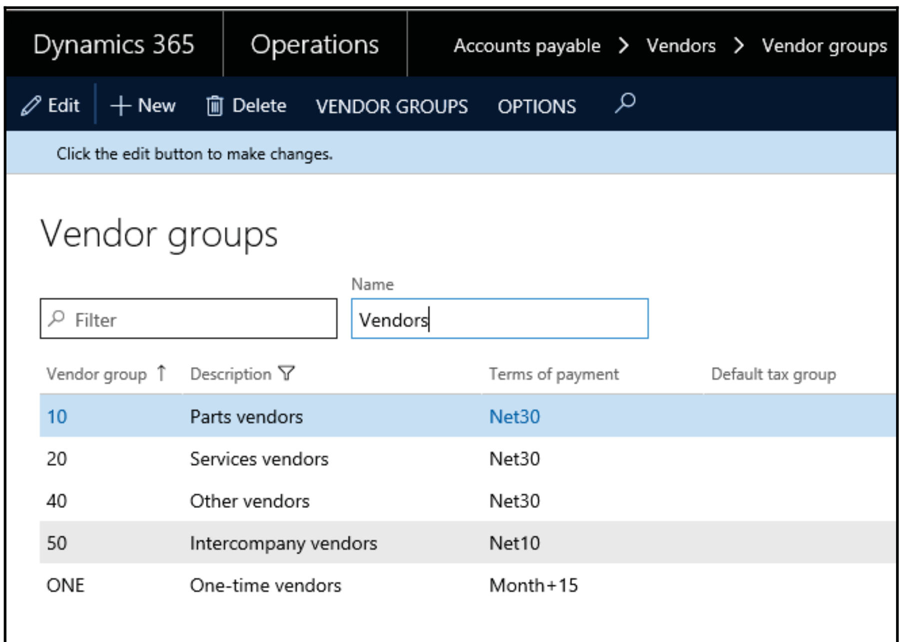
4. On the same control, override the control's `enter()` method with the following code snippet:

```
public void enter()
{
    super();
    this.setSelection(strLen(this.text()),
    strLen(this.text()));
}
```

5. Override the `executeQuery()` method of the `VendGroup` data source as follows:

```
public void executeQuery()
{
    QueryBuildRange qbrName;

    qbrName =
    SysQuery::findOrCreateRange(this.queryBuildDataSource(),
    fieldNum(VendGroup,Name));

    qbrName.value(FilterName.text() ?
     '*'+queryValue(FilterName.text())+'*' :
       SysQuery::valueUnlimited());

     super();
}
```

6. In order to test the search, build your solution, navigate to **Accounts payables** | **Vendors** | **Vendor groups,** and start typing in the **Name** filter. Note how the vendor group list is being filtered automatically:



How it works...

First of all, we add a new control, which represents the **Name** filter. Normally, the user's typing triggers the `textChange()` event method on the active control every time a character is entered. So, we override this method and add code to re-execute the form's query whenever a new character is typed in.

Next, we have to correct the cursor's behavior. Currently, once the user types in the first character, the search is executed and the system moves the focus out of this control and then moves back into the control selecting all the typed text. If the user continues typing, the existing text will be overwritten with the new character and the loop will continue.

In order to go around this, we have to override the control's `enter()` event method. This method is called every time the control receives a focus, whether it was done by a user's mouse, key, or by the system. Here, we call the `setSelection()` method. Normally, the purpose of this method is to mark a control's text, or a part of it, as selected. Its first argument specifies the beginning of the selection and the second one specifies the end. In this recipe, we use this method in a slightly different way. We pass the length of the typed text as a first argument, which means the selection starts at the end of the text. We pass the same value as a second argument, which means that selection ends at the end of the text. It does not make any sense from a selection point of view, but it ensures that the cursor always stays at the end of the typed text, allowing the user to continue typing.

The last thing to do is to add some code to the `executeQuery()` method to change the query before it is executed. Modifying the query was discussed in detail in the *Creating a custom filter control* recipe. The only thing to note here is that we add `*` to the beginning and the end of the search string to do the search with a partial string.

Note that the system's performance might be affected, as the data search is executed every time the user types in a character. It is not recommended to use this approach for large tables.

See also

- The *Creating a custom filter control* recipe

All your objects must belong to the same package, so you don't need to create a separate project for each recipe. You can add your code/object in your existing project/solution customization is possible only if the models are in the same package.

Building a selected/available list

Frequent users might note that some of the Dynamics 365 for Finance and Operations forms contain two sections placed next to each other and allow the moving of items from one side to the other. Normally, the right section contains a list of available values and the left one contains the values that have been chosen by the user. Buttons in the middle the allow the moving of data from one side to another. Double-click and drag and drop mouse events are also supported. Such design improves the user's experience, as data manipulation becomes more user-friendly.

Some of the examples in the standard application can be found at **General ledger** | **Chart of accounts** | **Dimensions** | **Financial dimension sets** or **System administration** | **Users** | **User groups**.

This functionality is based on the `SysListPanelRelationTableCallBack` application class. Developers only need to create its instance with the required parameters and the rest is done automatically.

This recipe will show the basic principles of how to create selected/available lists. We will add an option for assigning customers to buyer groups in the **Buyer groups** form in the **Inventory management** module.

How to do it...

Carry out the following steps in order to complete this recipe:

1. In the AOT, create a new table named `InventBuyerGroupList`. Let's not change any of its properties, as this table is for demonstration only.
2. Add a new field to the table with the following properties (click on **Yes** if asked to add a new relation to the table):

Property	**Value**
Type	String
Name	GroupId
ExtendedDataType	ItemBuyerGroupId

3. Add another field to the table with the following properties:

Property	**Value**
Type	String
Name	CustAccount
ExtendedDataType	CustAccount

4. In the AOT, open the `InventBuyerGroup` form and change its design's property as follows:

Property	Value
Style	Auto

5. Add a new `Tab` control with the following properties to the design's bottom:

Property	Value
Name	Tab
Width mode	SizeToAvailable
Height mode	Column height

6. Add a new `TabPage` control with the following properties to the newly created tab:

Property	Value
Name	BuyerGroups
Caption	Buyer groups

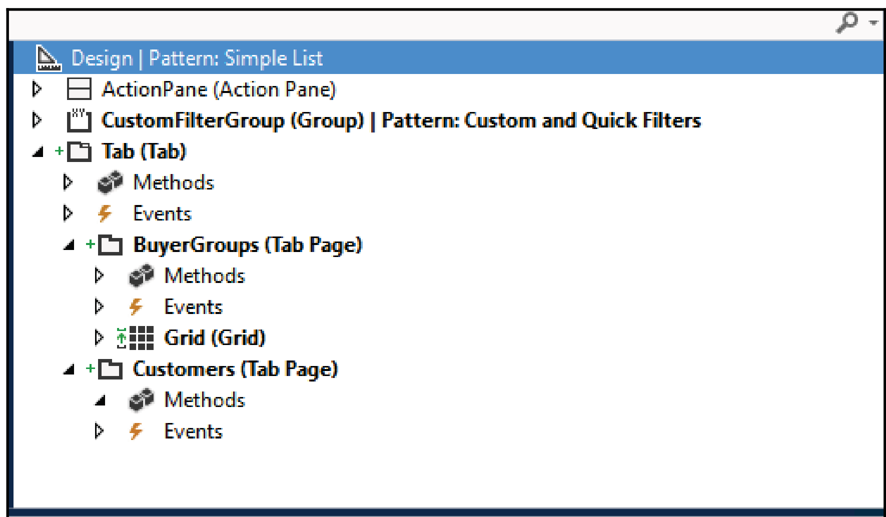
7. Add another `TabPage` control with the following properties to the newly created tab:

Property	Value
Name	Customers
Caption	Customers

8. Move the existing `Grid` control into the first tab page and disable the existing `CustomFilterGroup` group by setting its property:

Property	Value
Enable	No

9. The form will look similar to the following screenshot:



10. Add the following line to the form's class declaration:

```
SysListPanelRelationTable sysListPanel;
```

11. Override the form's `init()` method with the following code snippet:

```
public void init()
{
    container columns;
    #ResAppl

    columns = [fieldNum(CustTable, AccountNum)];


    sysListPanel = SysListPanelRelationTable::newForm(
    element,
    element.controlId(
    formControlStr(InventBuyerGroup,Customers)),
```

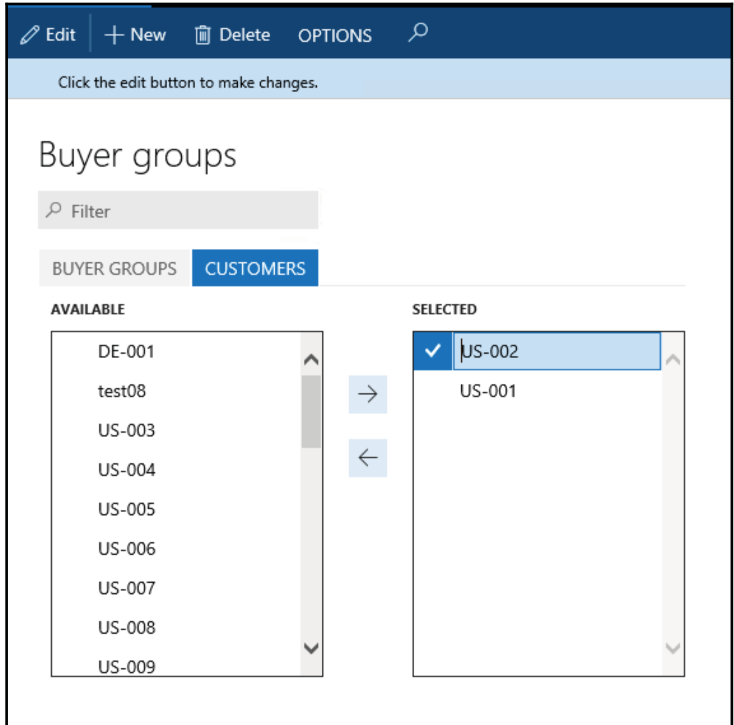
```
        "Selected",
        "Available",
        #ImageCustomer,
        tableNum(InventBuyerGroupList),
        fieldNum(InventBuyerGroupList,CustAccount),
        fieldNum(InventBuyerGroupList,GroupId),
        tableNum(CustTable),
        fieldNum(CustTable,AccountNum),
        columns);

        super();

        sysListPanel.init();

    }
```

12. Override the `pageActivated()` method on the newly created `Customers` tab page with the following code snippet:

```
public void pageActivated()
{
    sysListPanel.parmRelationRangeValue(
    InventBuyerGroup.Group);

    sysListPanel.parmRelationRangeRecId(
    InventBuyerGroup.RecId);

    sysListPanel.fill();

    super();
}
```

13. In order to test the list, first, save all your code and build your solution.

14. Now navigate to **Inventory management** | **Setup** | **Inventory** | **Buyer groups** and select any group. Then, go to the **Customers** tab page and use the buttons provided to move records from one side to the other. You can also double-click or drag and drop with your mouse:



How it works...

In this recipe, the `InventBuyerGroupList` table is used as a many-to-many relationship table between the buyer groups and the customers.

In terms of form design, the only thing that needs to be added is a new tab page. The rest is created dynamically by the `SysListPanelRelationTable` application class.

In the form's class declaration, we declare a new variable based on the `SysListPanelRelationTable` class and instantiate it in the form's `init()` method using its `newForm()` constructor. The method accepts the following parameters:

- The `FormRun` object representing the form itself.
- The name of the tab page.
- The label of the left section.
- The label of the right section.
- The number of the image that is shown next to each record in the lists.
- The relationship table number.
- The field number in the relationship table representing the child record. In our case, it is the customer account number--`CustAccount`.
- The field number in the relationship table representing the parent table. In this case, it is the buyer group number--`GroupId`.
- The number of the table that is displayed in the lists.
- A container of the field numbers displayed in each column.

We also have to initialize the list by calling its member method `init()` in the form's `init()` method right after the `super()` method.

The list's controls are created dynamically when the **Customers** tab page is opened. In order to accommodate that, we add the list's creation code to the `pageActivated()` event method of the newly created tab page. In this way, we ensure that the list is populated whenever a new buyer group is selected.

There's more...

The `SysListPanelRelationTable` class can only display fields from a single table. Alternatively, there is another application class named `SysListPanelRelationTableCallback`, which allows you to create more complex lists.

In order to demonstrate its capabilities, we will expand the previous example by displaying the customer name next to the account number. The customer name is stored in another table and can be retrieved by using the `name()` method on the `CustTable` table.

First, in the form's class declaration, we have to change the list declaration to the following code line:

```
SysListPanelRelationTableCallback sysListPanel;
```

Next, we create two new methods--one for the left list and the other one for the right list--that generate and return data containers to be displayed in each section. The methods will be placed on the `InventBuyerGroupList` table. In order to improve performance, these methods will be executed on the server tier (note the server modifier):

```
static server container selectedCustomers(
ItemBuyerGroupId _groupId)
{
   container            ret;
   container            data;
   CustTable            custTable;
   InventBuyerGroupList groupList;

   while select custTable
   order by AccountNum
   exists join groupList
   where groupList.CustAccount == custTable.AccountNum
   && groupList.GroupId     == _groupId

    {
       data = [custTable.AccountNum,
       custTable.AccountNum,
       custTable.name()];

       ret += [data];
    }

      return ret;
}

    static server container availableCustomers(
    ItemBuyerGroupId _groupId)
  {
     container            ret;
     container            data;
     CustTable            custTable;
     InventBuyerGroupList groupList;

     while select custTable
     order by AccountNum
     notExists join firstOnly groupList
     where groupList.CustAccount == custTable.AccountNum
```

```
 && groupList.GroupId     == _groupId
{
   data = [custTable.AccountNum, custTable.AccountNum,
   custTable.name()];

   ret += [data];
}

   return ret;
}
```

Each of the methods returns a container of containers. The outer container holds all the items in the list. The inner container represents one item in the section and it contains three elements--the first is the identification number of the element and the next two are displayed on the screen.

Next, we create two new methods with the same names on the `InventBuyerGroup` form itself. These methods are required to be present on the form by the `SysListPanelRelationTableCallback` class. These methods are nothing but wrappers to the previously created methods:

```
private container selectedCustomers()
{
   return InventBuyerGroupList::selectedCustomers(
   InventBuyerGroup.Group);
}

private container availableCustomers()
{
   return InventBuyerGroupList::availableCustomers(
   InventBuyerGroup.Group);
}
```

In this way, we reduce the number of calls between the client and server tiers while generating the lists.

Finally, we replace the form's `init()` method with the following code snippet:

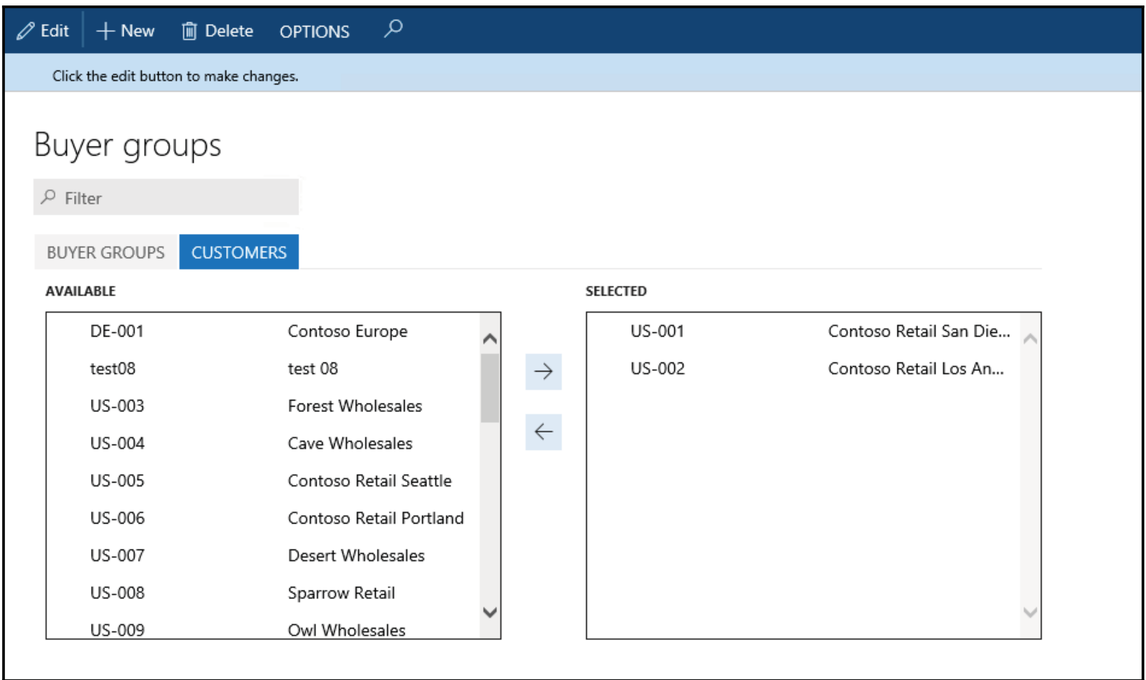
```
public void init()
{
    container columns;
    #ResAppl

    columns = [0, 0];

    sysListPanel = SysListPanelRelationTableCallback::newForm(
    element,element.controlId(
```

```
            formControlStr(InventBuyerGroup,Customers)),
            "Selected",
            "Available",
            #ImageCustomer,
            tableNum(InventBuyerGroupList),
            fieldNum(InventBuyerGroupList,CustAccount),
            fieldNum(InventBuyerGroupList,GroupId),
            tableNum(CustTable),
            fieldNum(CustTable,AccountNum),
            columns,
            0,
            '',
            '',
            identifierStr(selectedCustomers),
            identifierStr(availableCustomers));

            super();

            sysListPanel.init();

        }
```

This time, we used the `newForm()` constructor of the `SysListPanelRelationTableCallback` class, which is very similar to the previous one, but accepts the names of methods as arguments, which will be used to populate the data in the right and left sections.

Note that the `columns` container that previously held a list of fields now contains two zeros. By doing that, we simply define that there will be two columns in each list. Since the lists are actually generated outside the `SysListPanelRelationTableCallback` class, we do not need to specify the field numbers of the columns anymore.

Now, when you run the **Buyer groups** form, both sections contain a **customer name** column:



Creating a wizard

Wizards in Dynamics 365 for Finance and Operations are used to help a user to perform a specific task. An example of a standard Dynamics 365 for Finance and Operations wizards is the **Number Sequence Wizard**.

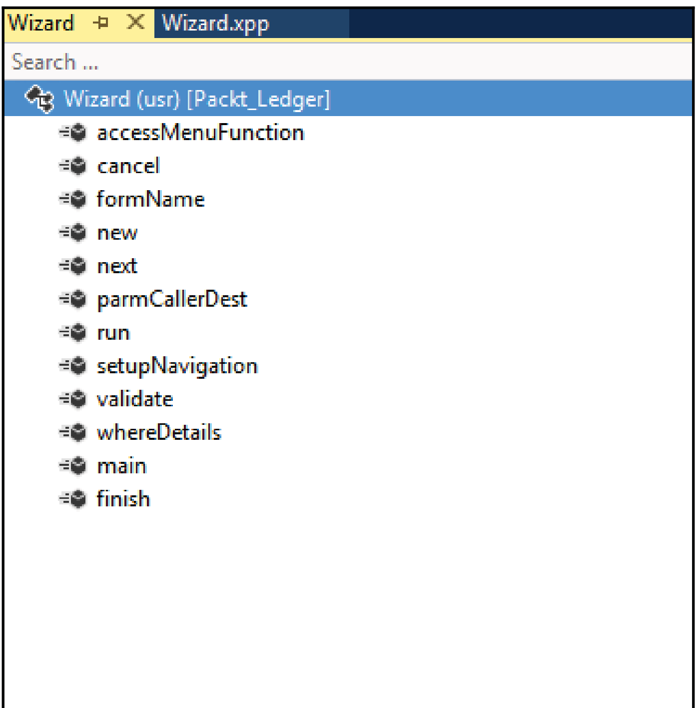
Normally, a wizard is presented to a user as a form with a series of steps. During the wizard run, all the user's inputs are collected and committed to the database. Then, the user presses the **Finish** button on the last wizard page.

In this recipe, we will create a new wizard to create main accounts. First, we will use the standard Dynamics 365 for Finance and Operations wizard to create a framework, and then we will add some additional controls manually.

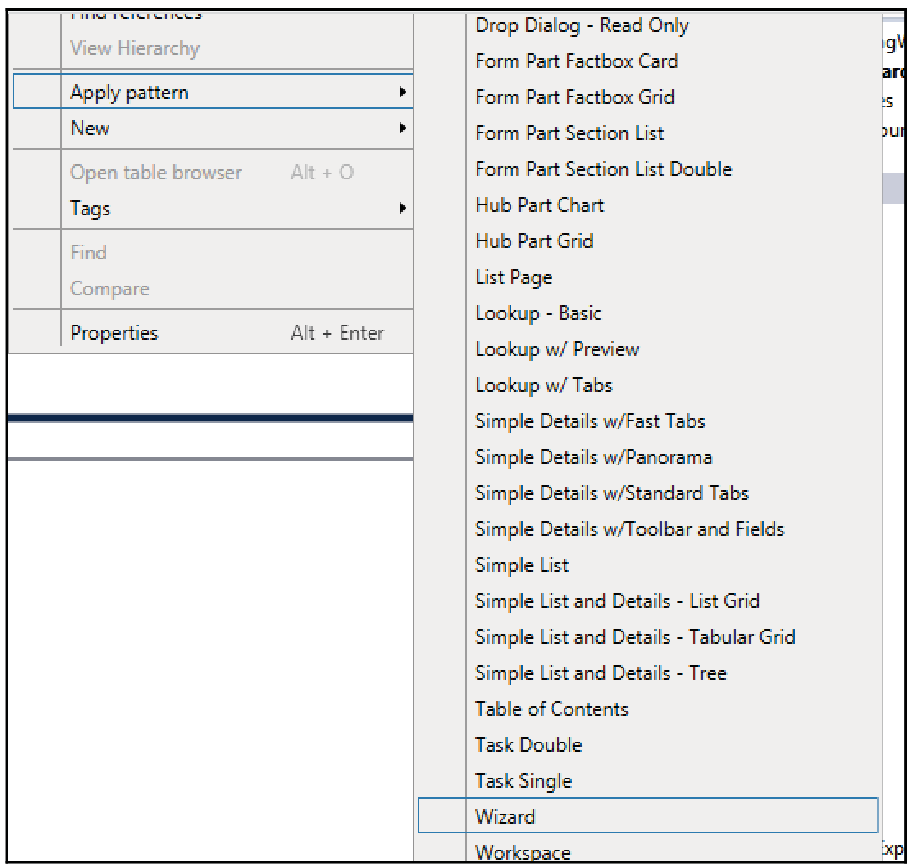
How to do it...

Carry out the following steps in order to complete this recipe:

1. In the Development Workspace, create a new Dynamics 365 for Operations project.
2. Create a new **Class** named `wizard` that extends `SysWizard`:



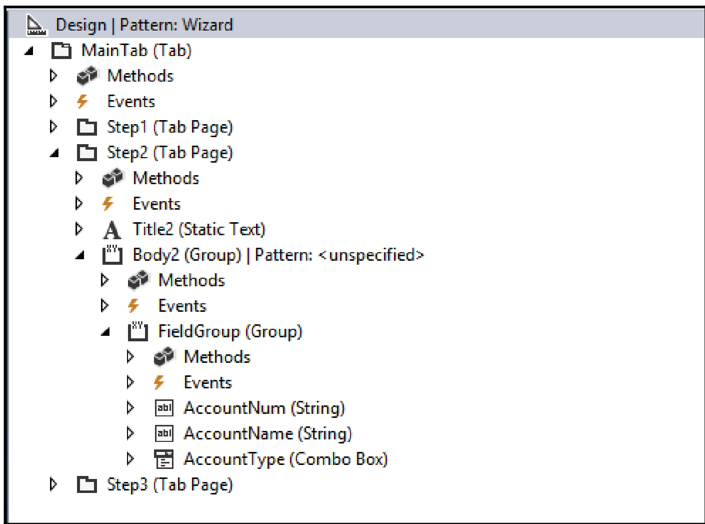
3. Create a new **Form** named `wizard`, select a design, and apply the design pattern `Wizard`:



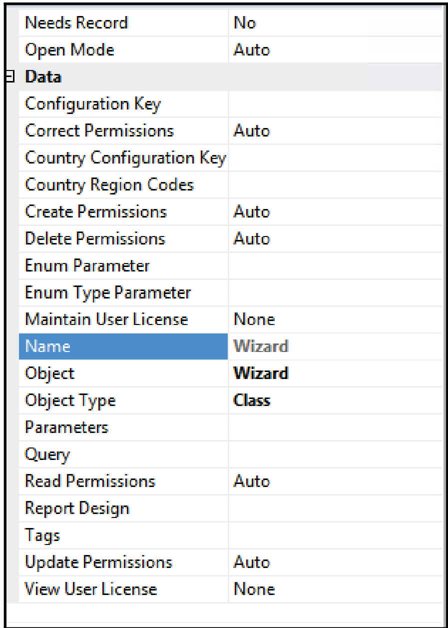
4. Address BP Warnings:

- `Design.Caption` isn't empty
- The form must be referenced by at least one menu item
- `TabPage.Caption` isn't empty (for all wizard content pages)
- `MainInstruction.Text` isn't empty (for all wizard content pages)

The form design should look as follows:



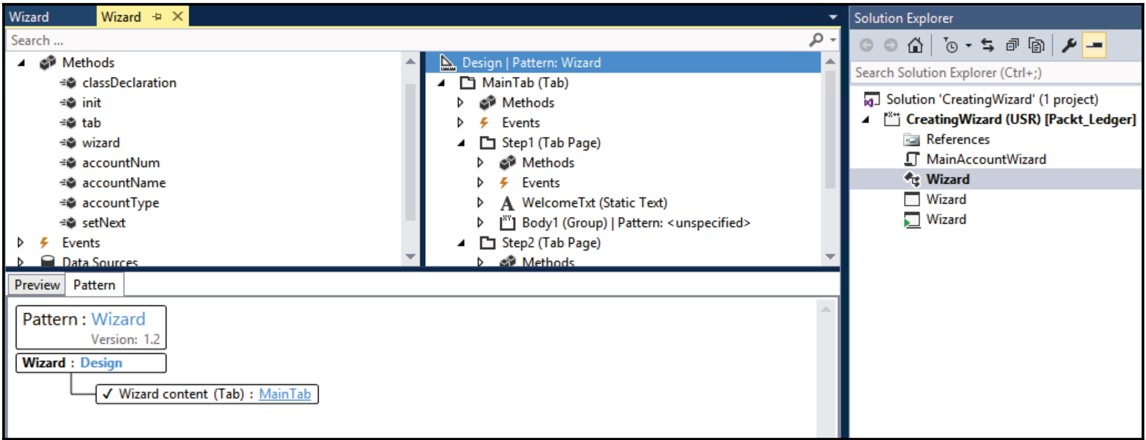
5. Create a new display menu item named `Wizard`, set its **Object Type** as `Class` and **Object** as `Wizard`, and the properties window should look as follows:



Needs Record	No
Open Mode	Auto
Data	
Configuration Key	
Correct Permissions	Auto
Country Configuration Key	
Country Region Codes	
Create Permissions	Auto
Delete Permissions	Auto
Enum Parameter	
Enum Type Parameter	
Maintain User License	None
Name	Wizard
Object	**Wizard**
Object Type	**Class**
Parameters	
Query	
Read Permissions	Auto
Report Design	
Tags	
Update Permissions	Auto
View User License	None

6. Create a new macro library named `MainAccountWizard` with the following line of code:

```
#define.tabStep2(2)
```



7. Modify the `Wizard` class by adding the following lines of code to its class declaration:

```
MainAccount mainAccount;
#MainAccountWizard
```

8. Add a new method in the class name `accessMenuFunction`:

```
/// <summary>
/// Retrieves a menu function.
/// </summary>
/// <returns>
/// The menu function.
/// </returns>
public MenuFunction accessMenuFunction()
{
    return new
    MenuFunction
     (menuitemDisplayStr(Wizard),MenuItemType::Display);
}
```

9. Override the method `formname()` and add the following line of code:

```
return formStr(Wizard);
```

10. Add a new method `parmCallerDest` and use the following code:

```
/// <summary>
/// Gets or sets caller destination (used to get parameter from
menu item)
/// </summary>
/// <param name="_callerDest">Caller destination.</param>
/// <returns>Caller destination</returns>
public str parmCallerDest(str _callerDest = callerDest)
{
    callerDest = _callerDest;

    return callerDest;
}
```

11. Add the following line of code to the overridden method `setupNavigation()` in the same class:

```
nextEnabled[#tabStep2] = false;
```

12. Override the `finish()` method of the class with the following code snippet:

```
protected void finish()
{
    mainAccount.initValue();
    mainAccount.LedgerChartOfAccounts =
    LedgerChartOfAccounts::current();
    mainAccount.MainAccountId = formRun.accountNum();
    mainAccount.Name = formRun.accountName();
    mainAccount.Type = formRun.accountType();

    super();
}
```

13. Replace the `validate()` method of the same class with the following code snippet:

```
boolean validate()
{
    return mainAccount.validateWrite();
}
```

14. Replace the `run()` method of the same class with the following code snippet:

```
 void run()
{
    mainAccount.insert();

    info(strFmt(
    "Ledger account '%1' was successfully created",
    mainAccount.MainAccountId));
}
```

15. In the `Wizard` form, add the following line of code to its class declaration:

```
 #MainAccountWizard
```

16. Change the form's design property:

Property	Value
Caption	Main account wizard

17. Modify the properties of the `Step1` tab page, as follows:

Property	Value
Caption	Welcome

18. Add a new `StaticText` control in this tab page with the following properties:

Property	Value
Name	WelcomeTxt
Text	This wizard helps you to create a new main account.

19. Modify the properties of the `Step2` tab page:

Property	Value
Caption	Account setup
HelpText	Specify account number, name, and type.

20. Add a new `StringEdit` control in this tab page with the following properties:

Property	Value
Name	AccountNum
AutoDeclaration	Yes
Label	Main account
ExtendedDataType	AccountNum

21. Add one more `StringEdit` control in this tab page with the following properties:

Property	Value
Name	AccountName
AutoDeclaration	Yes
ExtendedDataType	AccountName

22. Add a new `ComboBox` control in this tab page with the following properties:

Property	Value
Name	AccountType
AutoDeclaration	Yes
EnumType	DimensionLedgerAccountType

23. Modify the properties of the `Step3` tab page, as follows:

Property	Value
Caption	Finish

24. Add a new `StaticText` control on this tab page with the following properties:

Property	Value
Name	FinishTxt
Text	This wizard is now ready to create new main account.

25. Create the following four methods at the top level of the form:

```
public MainAccountNum accountNum()
{
   return AccountNum.text();
}

public AccountName accountName()
{
   return AccountName.text();
}

public DimensionLedgerAccountType accountType()
{
   return AccountType.selection();
}
public void setNext()
{
   sysWizard.nextEnabled(
   this.accountNum() && this.accountName(),
   #tabStep2,
   false);
}
```

26. Add the following code in the `init()` method of the form:

```
 public void init()
{
  super();
  if (element.Args().caller())
  {
    sysWizard = element.Args().caller();
  }
  else
  {
    Wizard::main(new Args());
    element.closeCancel();
  }
}
```

27. Add a new method named `tab()` and place the following code:

```
 FormTabControl tab()
{
   return MainTab;
}
```

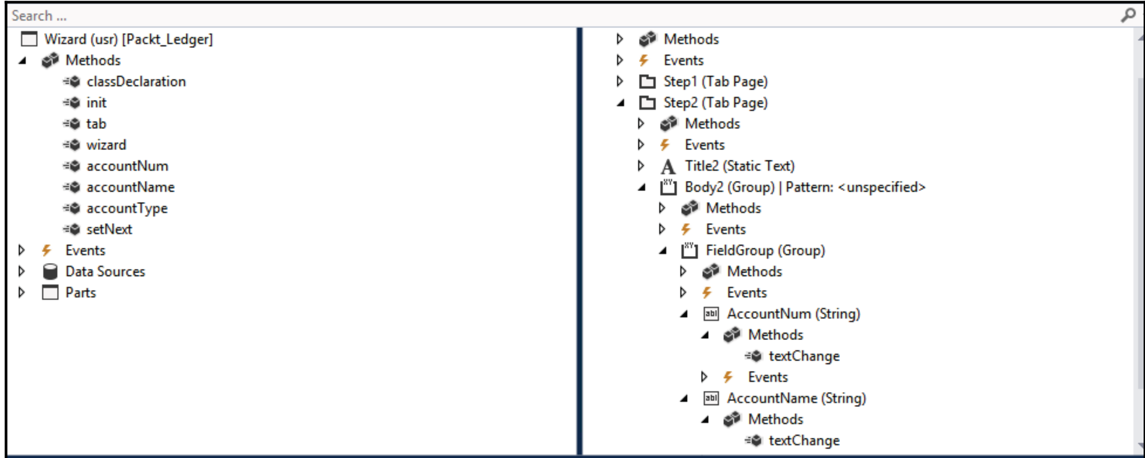
28. Add a new method that returns the reference of the wizard instance:

```
  SysWizard wizard()
{
   return sysWizard;
}
```

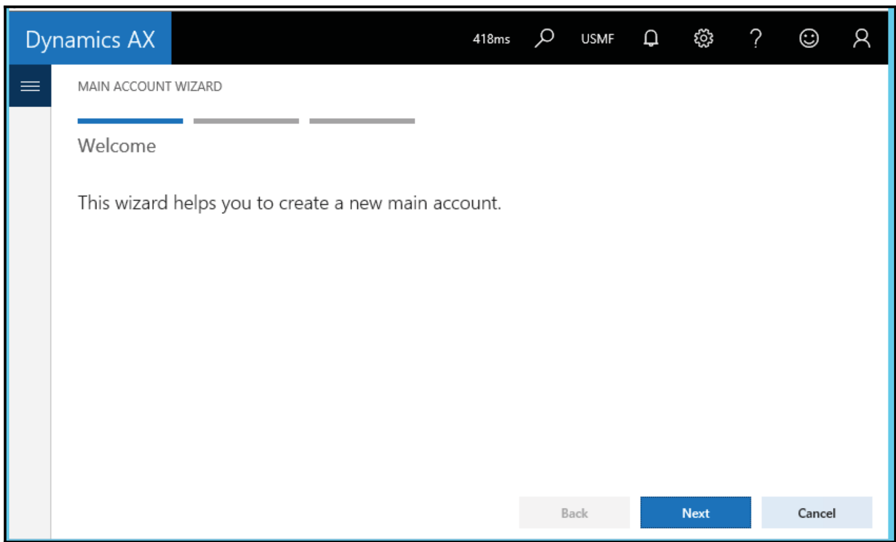
29. Now, override the `textChange()` method on the `AccountNum` and `AccountName` controls with the following code:

```
public void textChange()
{
   super();
   element.setNext();
}
```

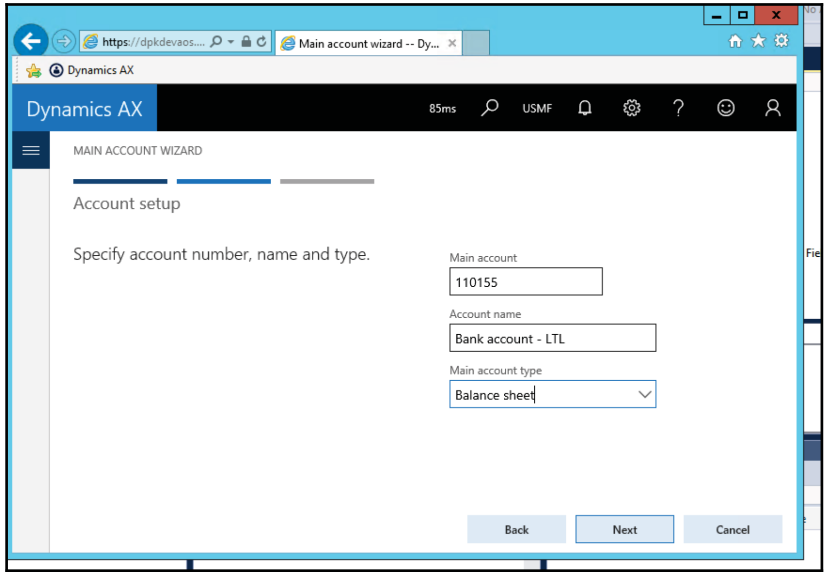
After all modifications, the form will look as follows:



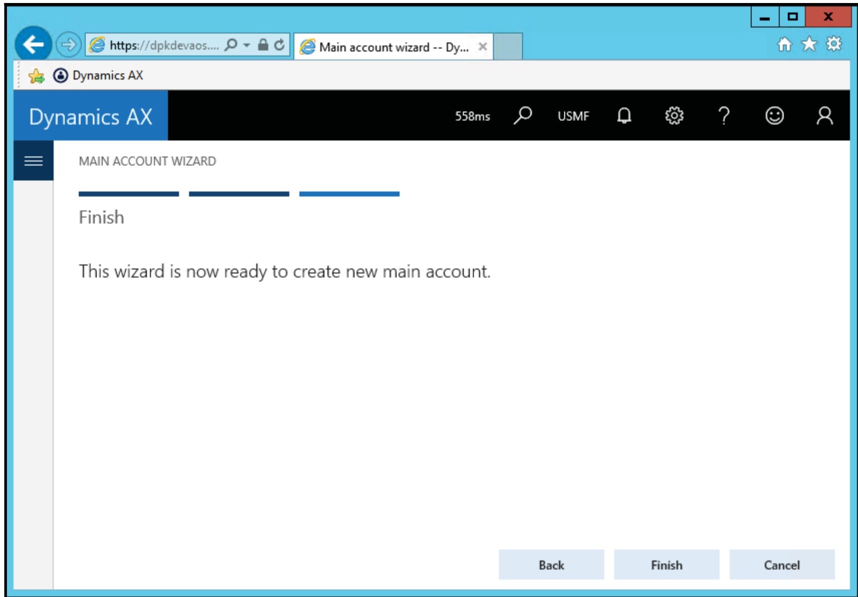
30. In order to test the newly created wizard, run the **Wizard** menu item, and the wizard will appear. On the first page, click on **Next**:



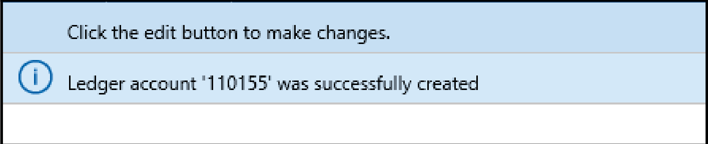
31. On the second page, specify **Main account**, **Account name**, and **Main account type**:



32. On the last page, click on **Finish** to complete the wizard:



33. The **Infolog** window will display a message that a new account was created successfully:



How it works...

The wizard creates three AOT objects for us:

- The `Wizard` class, which contains all the logic required to run the wizard
- The `Wizard` form, which is the wizard layout
- Finally, the `Wizard` display menu item, which is used to start the wizard and can be added to a menu

The generated wizard is just a starting point for our custom wizard. It already has three pages, as we specified during its creation, but we still have to add new user input controls and custom code in order to implement our requirements.

We start by defining a new `#tabStep2` macro, which holds the number of the second tab page. We are going to refer to this page several times, so it is good practice to define its number in one place.

In the `Wizard` class, we override its `setupNavigation()` method, which is used to define initial button states. We use this method to disable the **Next** button on the second page by default. The `nextEnabled` variable is an array holding the initial enabled or disabled state for each tab page.

The overridden `finish()` method is called when the user clicks on the **Finish** button. Here, we initialize the record and then assign user input to the corresponding field values.

In the `validate()` method, we check the account that will be created. This method is called right after the user clicks on the **Finish** button at the end of the wizard and before the main code is executed in the `run()` method. Here, we simply call the `validateWrite()` method for the record from the main account table.

The last thing to do in the class is to place the main wizard code--insert the record and display a message--in the `run()` method.

In the `Wizard` form's design, we modify properties of each tab page and add text to explain to the user the purpose of each step. Note that the `HelpText` property value on the second tab page appears as a step description right below the step title during runtime. This is done automatically by the `SysWizard` class.

Finally, on the second tab page, we place three controls for user input. Later on, we create three methods which return the controls' values: account number, name, and type values, respectively. We also override the `textChange()` event methods on the controls to determine and update the runtime state of the **Next** button. These methods call the `setNext()` method, which actually controls the behavior of the **Next** button. In our case, we enable the **Next** button as soon as all input controls have values.

Processing multiple records

In Dynamics 365 for Finance and Operations, by default, most of the functions available on forms are related to currently selected single record. However, on many Dynamics 365 for Finance and Operations forms you will find a multiple record selection option, but at the same time ,you will be able to perform some certain operations only. So, to perform a specific operation on all selected records, some modification is required.

In this recipe, we will explore how to process multiple records at the same time. You can modify an existing process for the same. For this demonstration, we will add a new button to the action pane on the **Vend Table** form to show multiple selected accounts in the **Infolog** window.

How to do it...

For this recipe, we will extend the `VendTable` form. Currently, we don't have an option to put multiple vendors on hold in a single click. So, we will add a new button there to process all selected vendors. Carry out the following steps in order to complete this recipe:

Add a new project in your Visual Studio Solution `ProcessingMultipleRecords`, create an extension of the `VendTable` form from AOT, and add this to the project.

1. To process selected records, add a new button `ProcessSelected` to the `VendorModifyButtonGroup` control in the action pane. Add the button here with the following properties:

Property	Value
Name	ProcessSelected
Text	On hold (All selected)
MultiSelect	Yes

2. Create a new extension class `VendTableFrom_Extension` for the `VendTable` form and add a method with the following code snippet:

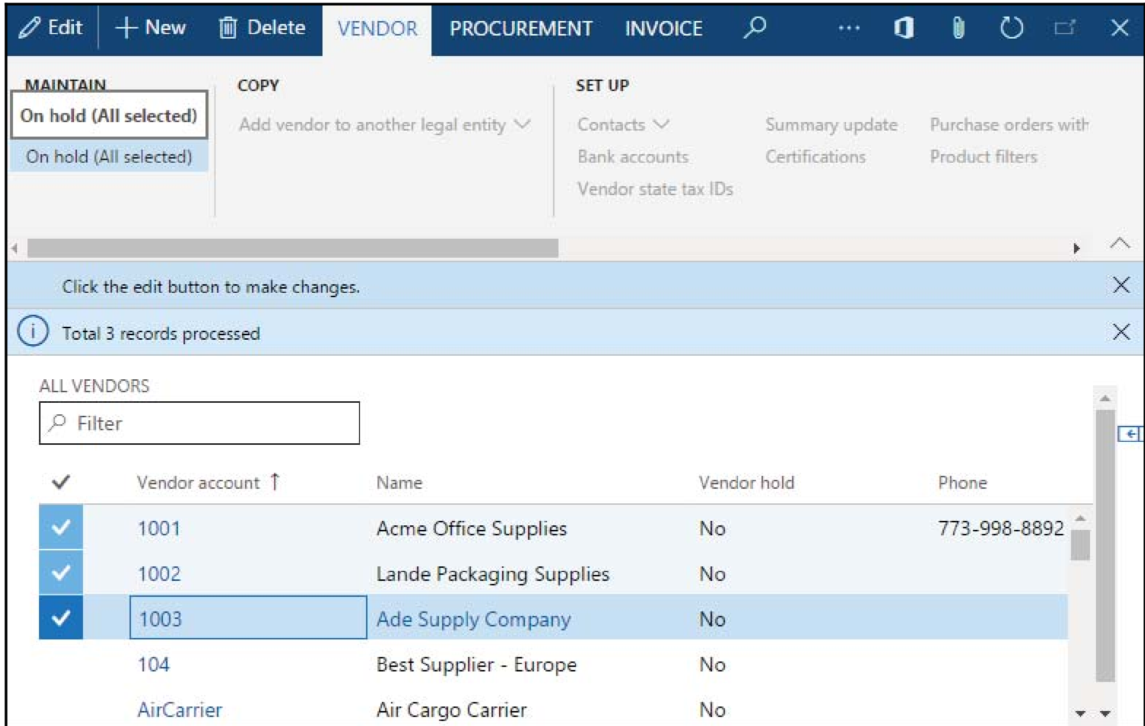
```
[ExtensionOf(formstr(VendTable))]
final class VendTableFrom_Extension
public void processSelected(FormControl sender,
FormControlEventArgs e)
{
  VendTable tmpVendTable, updateVendTable;
```

```
    int      recordUpdated;
    for(tmpVendTable = this.VendTable_ds.getFirst(true) ?
    this.VendTable_ds.getFirst(true) :
    this.VendTable_ds.cursor(); tmpVendTable; tmpVendTable =
    this.VendTable_ds.getNext())
   {
      ttsbegin;
      select firstonly forupdate updateVendTable where
      updateVendTable.AccountNum == tmpVendTable.AccountNum;
      updateVendTable.Blocked = CustVendorBlocked::All;
      updateVendTable.update();
      recordUpdated++;
      ttscommit;
   }
      info(strFmt("Total %1 records processed", recordUpdated));
 }
}
```

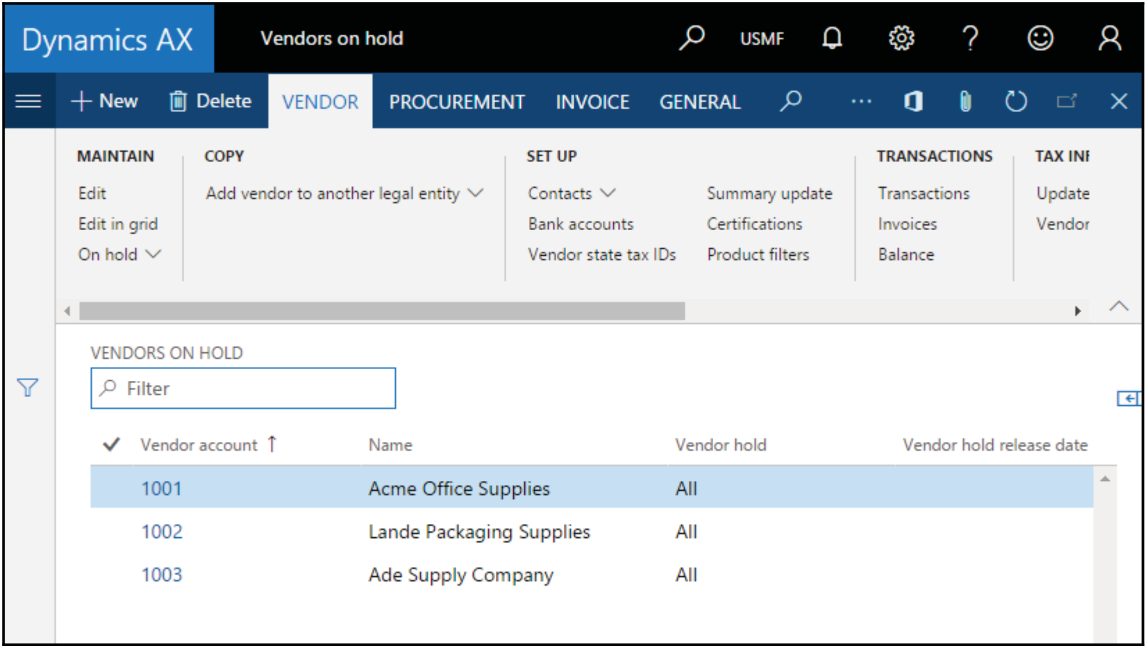
3. Now, copy the `OnClicked` event handler from the `ProcessSelected` button control and add a new method in `VendTableFrom_Extension` with the following code snippet:

```
/// <summary>
///
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
[FormControlEventHandler(formControlStr(VendTable,
ProcessSelected), FormControlEventType::Clicked)]
public void ProcessSelected_OnClicked(FormControl sender,
FormControlEventArgs e)
{
   this.processSelected(sender,e);
}
```

4. In order to test the record selection, navigate to **Accounts payable** | **Vendor** | **All vendors**, select several records, and click on the new **On Hold (All selected)** button.



5. The selected items will be displayed in the Vendors on hold form as well. To check, navigate to **Accounts payable** | **Vendor** | **Vendors on hold.**



How it works...

In earlier versions, we had `MultiSelectionHelper` to support such customization. In current versions, this class has been deprecated. So we have to travel record by record here.

Next, get the first marked record using `VendTable_DS.getFirst(true)`, and then go through all the other marked records (if any) using `VendTable_DS.getnext()` and process them one by one. In this demonstration, we simply put vendors on hold using this code.

The last thing to do is to update the table using another object of the table. Note that the button's `MultiSelect` property is set to `Yes` to ensure it is still enabled when multiple records are marked.

Coloring records

One of Dynamics 365 for Operation's exciting features, which can enhance user experience, is the ability to color individual records. Some users might find the system more intuitive and user-friendly through this modification.

For example, emphasizing the importance of disabled records by highlighting terminated employees or former customers in red allows users to identify relevant records at a glance. Another example is to show processed records, such as posted journals or invoiced sales orders, in green.

Getting ready

In this recipe, we will learn how to change a record's color. We will use one created earlier form the `PktDisabledUser` form located in **System administration** | **Users** | **Disabled Users with color** and add a method to show disabled users in red.

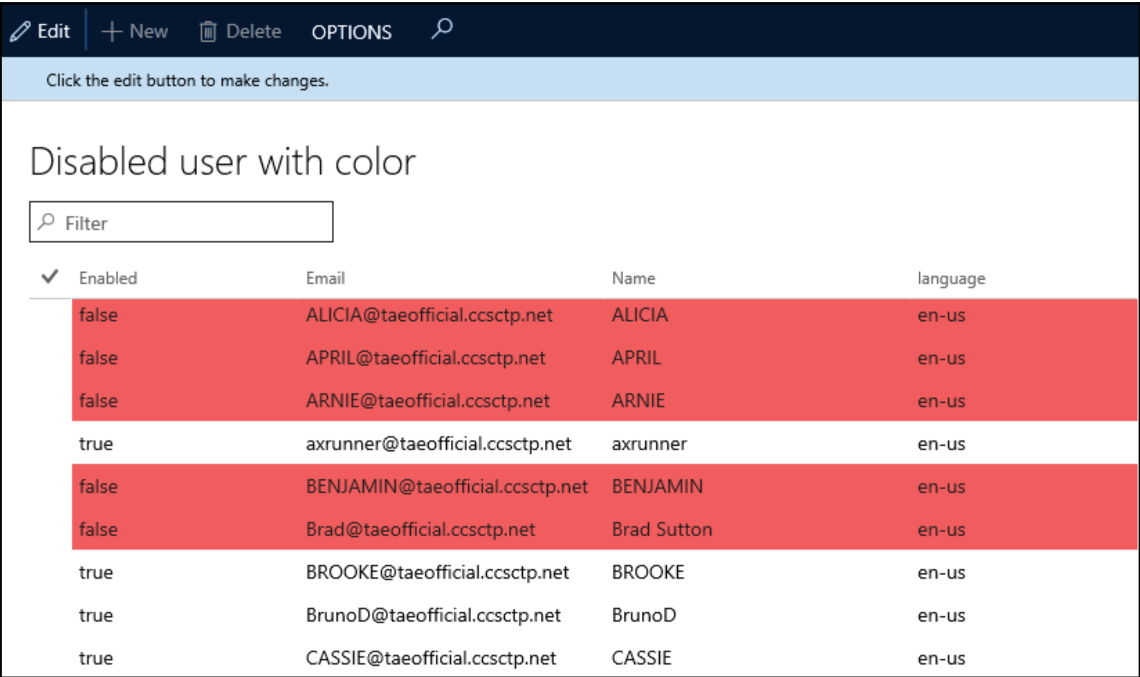
How to do it...

1. Add a new project in your solution.
2. Go to application explorer and search for the `PktDisabledUser` form; add this form to your project.
3. Now, override the `displayOption()` method in its **UserInfo** data source with the following code snippet:

```
 public void displayOption(
 Common _record,
 FormRowDisplayOption _options)
{
   if (!_record.(fieldNum(UserInfo,Enable)))
  {
     _options.backColor(WinAPI::RGB2int(255,100,100));
  }

 super(_record, _options);
}
```

You should take care with the selected model for your project, every time you add/customize an object into your project.

4. In order to test the coloring, navigate to **System administration** | **Users** | **Users** | **Disabled Users with color** and note how disabled users are now displayed in a different color:



How it works...

The `displayOption()` method on any form's data source can be used to change some of the visual options. Before displaying each record, this method is called by the system with two arguments--the first is the current record and the second is a `FormRowDisplayOption` object--whose properties can be used to change a record's visual settings just before it appears onscreen. In this example, we check whether the current user is disabled, and if they are, we change the background property to light red by calling the `backColor()` method with the color code.

In this example, we used the `_record.(fieldNum(UserInfo,Enable))` expression to address the `Enable` field on the `UserInfo` table. This type of expression is normally used when we know the type of record, but it is declared as a generic `Common` type.

For demonstration purposes, we specified the color directly in the code, but it is a good practice for the color code to come from a configuration table. See the *Creating a color picker lookup* recipe in `Chapter 4`, *Building Lookups*, to learn how to allow the user to choose and store the color selection.

See also

- The *Creating a color picker lookup* recipe in `Chapter 4`, *Building Lookups*
- The *Creating a new form* recipe in `Chapter 2`, *Working with forms*

Adding an image to records

Company-specific images in Dynamics 365 for Finance and Operations can be stored along with the data in the database tables. They can be used for different purposes, such as a company logo that is displayed in every printed document, employee photos, inventory pictures, and so on.

Images are binary objects and they can be stored in the container table fields. In order to make the system perform better, it is always recommended to store images in a separate table so that it does not affect the retrieval speed of the main data.

One of the most convenient ways to attach images to record is to use the **Document handling** feature of Dynamics 365 for Finance and Operations. It does not require any change in the application. However, the **Document handling** feature is a very generic way of attaching files to record and might not be suitable for specific circumstances.

Another way of attaching images to records is to utilize the standard application objects, though minor application changes are required. For example, the company logo in the **Legal entities** form, located at **Organization administration** | **Setup** | **Organization**, is one of the places where the images are stored that way.

In this recipe, we will explore the latter option. As an example, we will add the ability to store an image for each customer. We will also add a new **Image** button on the **Customers** form, allowing us to attach or remove images from customers.

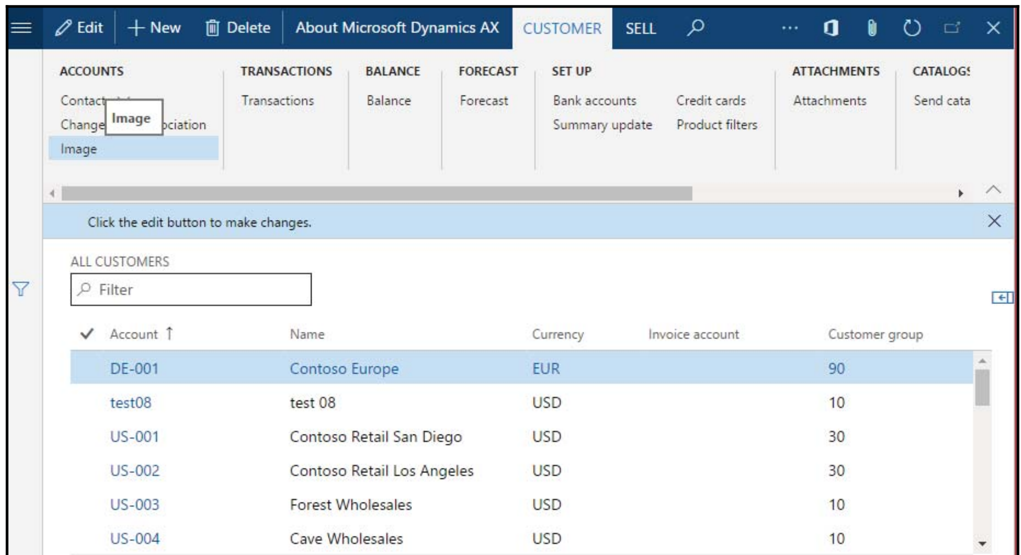
How to do it...

Carry out the following steps in order to complete this recipe:

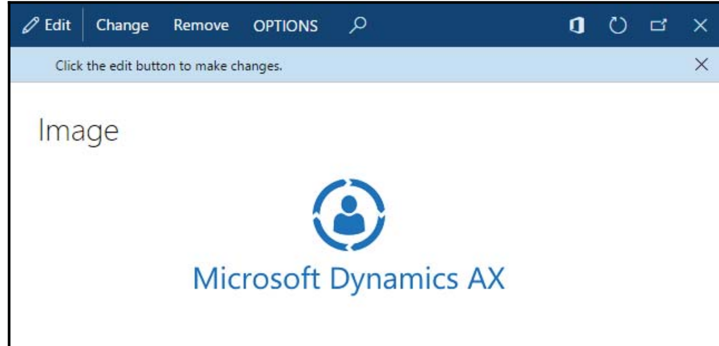
1. Open the `CustTable` form in the AOT. Add a new `MenuItemButton` control at the bottom of the **Accounts** button group, which is located at **ActionPaneHeader** | **aptabCustomer** | **btngrpCustomerAccounts**, with the following properties:

Property	Value
Name	Image
Text	Image
ButtonDisplay	TextWithImageAbove
NormalImage	10598
ImageLocation	EmbeddedResource
DataSource	CustTable
MenuItemType	Display
MenuItemName	CompanyImage

2. Navigate to **Accounts receivable** | **Customers** | **All customers** and note the new **Image** button in the action pane:



3. Click on the button, and then use the **Change** button to upload a new image for the selected item:



- The **Remove** button can be used to delete an existing image.

How it works...

In this demonstration, there are only three standard Dynamics 365 for Finance and Operations objects used:

- The `CompanyImage` table, which holds image data and information about the record to which the image is attached. The separate table allows you to easily hook image functionality to any other existing table without modifying that table or decreasing its performance.
- The `CompanyImage` form, which shows an image and allows you to modify it.
- The `Display` menu item `CompanyImage`, which allows you to open the form.

We added the menu item to the `CustTable` form and modified some of its visual properties. This ensures that it looks consistent with the rest of the action pane. We also changed its `DataSource` property to the `CustTable` data source. This makes sure that the image is stored against that record.

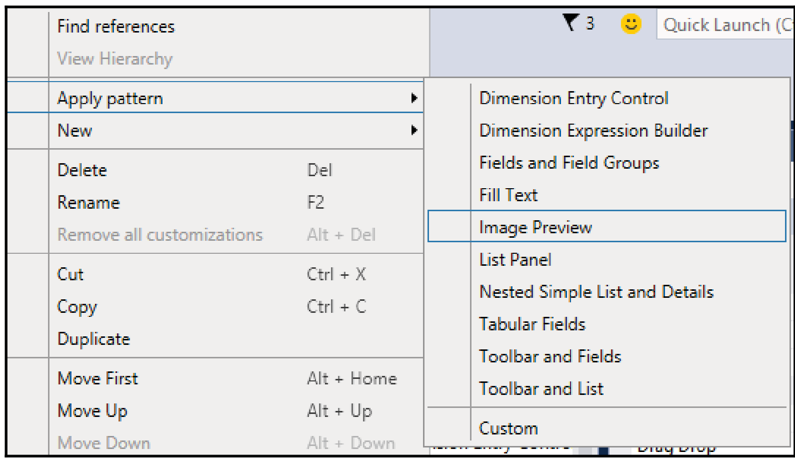
There's more...

The following two topics will explain how a stored image can be displayed as a new tab page on the main form and how it can be saved back to a file.

Displaying an image as part of a form

In this section, we will extend the recipe by displaying the stored image on a new tab page on the **Customers** form.

Firstly, we need to add a new tab page to the end of the `CustTable` form's `TabHeader` control, which is located inside another tab page called `TabPageDetails`. This is where our image will be displayed. Right-click on this new tab and set the patterns to `Image Preview` as follows:



Set the following properties of the new tab page:

Property	Value
Name	TabImage
AutoDeclaration	Yes
Caption	Image

Add a new `Window` type control to the tab page. This control will be used to display the image. Set its properties as follows:

Property	Value
Name	CustImage
AutoDeclaration	Yes

Next, let's create a new method at the top level of the `CustTable` form:

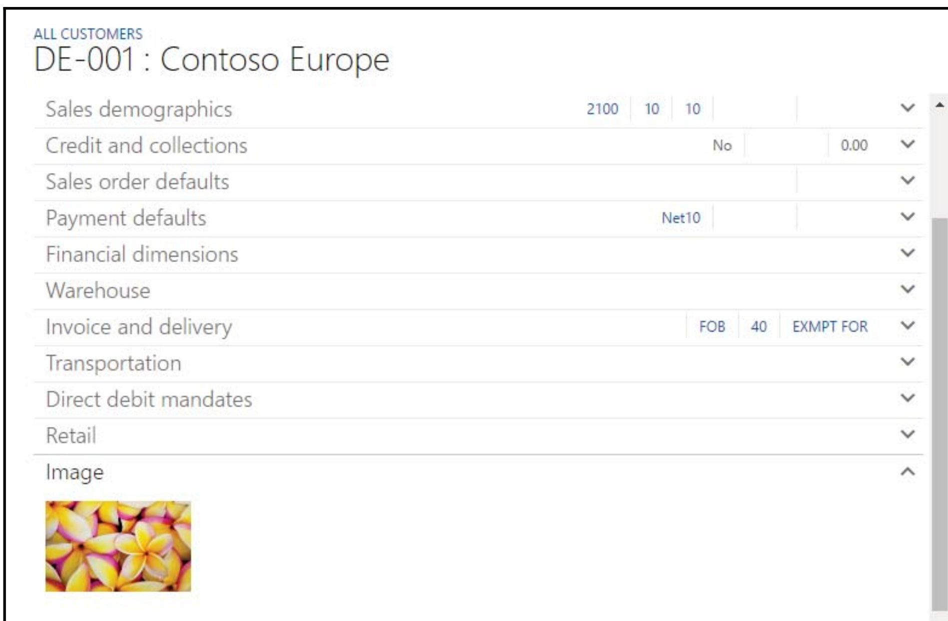
```
public void loadImage()
{
   Image        img;
   CompanyImage companyImage;

   companyImage = CompanyImage::find(
   CustTable.dataAreaId,
   CustTable.TableId,
   CustTable.RecId);

   if (companyImage.Image)
  {
     img = new Image();
     img.setData(companyImage.Image);
     CustImage.image(img);
  }
   else
  {
     CustImage.image(null);
  }

}
```

This method finds a `CompanyImage` record first, which is attached to the current record, and then displays the binary data using the `CustImage` control. If no image is attached, the `Window` control is cleared to display an empty space.

Next, we add the following line of code after super to the bottom of the `selectionChanged()` method of the `CustTable` data source to ensure that the image is loaded for a currently selected record:

```
element.loadImage();
```

Now, save all your code and build the VS solution. To test your code, navigate to **Account receivable** | **Customers** | **All customers**, select previously used customers, and click on the customer **Account** number in the grid. On the **Customers** form, note the new tab page with the image displayed:



Saving a stored image as a file

This section will describe how the stored image can be restored back to a file. This is quite a common case when the original image file is lost. We will enhance the standard `Image` form by adding a new **Save as** button, which allows us to save the stored image as a file.

Let's find the `CompanyImage` form in the AOT and add a new `Button` control to the form's `ButtonGroup`, which is located in the first tab of the `ActionPane` control. Set the button's properties as follows:

Property	Value
Name	SaveAs
Text	Save as

Create a new method at the top level of the form:

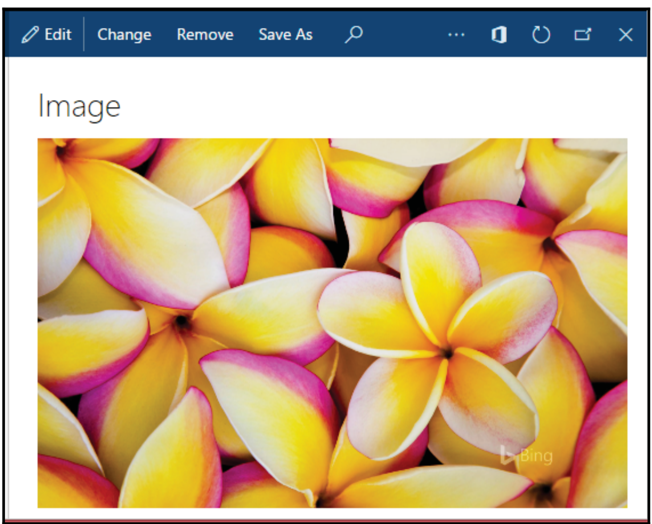
```
public void saveImage()
{
    Image    img;
    Filename name;
    str      type;
    #File

    if (!imageContainer)
    {
        return;
    }

    img = new Image();
    img.setData(imageContainer);

    type = '.'+strLwr(enum2value(img.saveType()));
    name = WinAPI::getSaveFileName(
    element.hWnd(),
    [WinAPI::fileType(type),#AllFilesName+type],
    '',
    '');

    if (name)
    {
        img.saveImage(name);
    }
}
```

This method will present the user with the **Save as** dialog, allowing them to choose the desired filename to save the current image. Note that the `imageContainer` form variable holds image data. If it is empty, it means there is no image attached, and we do not run any of the code. We also determine the loaded file type to make sure our **Save as** dialog shows only files of that particular type, for example, JPEG.

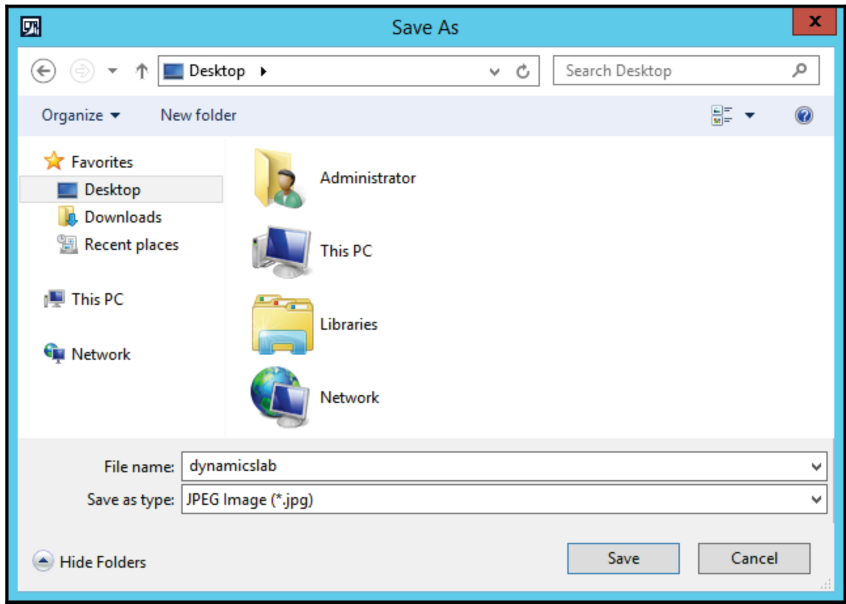
Override the button's `clicked()` method with the following code snippet to make sure that the `saveImage()` method is executed once the user clicks on the button:

```
void clicked()
{
    super();
    element.saveImage();
}
```

Now, save all your changes and build your solution. To test, go to **Account receivable** | **Customers** | **All customers**, click on the **Image** button, and you will find that a new **Save as** button is available:



Use this button to save the stored image as a file:



Note that the `CompanyImage` form is used system-wide and the new button is available across the whole system now.

We do not recommend overlaying unless it's the only option. We urge you to always try to use extensions, event handlers, and events for your development to achieve any requirement rather than overlaying standard objects. You may have found some recipes here with overlaying objects; all of them just for illustration purposes, and to simplify the explanation of the actual agenda of the recipe.

4

Building Lookups

In this chapter, we will cover the following recipes:

- Creating an automatic lookup
- Creating a lookup dynamically
- Using a form to build a lookup
- Building a tree lookup
- Displaying a list of custom options
- Displaying custom options in another way
- Building a lookup based on the record description
- Building the browse for folder lookup
- Creating a color picker lookup

Introduction

Lookups are the standard way to display a list of possible selection values to the user, while editing or creating database records. Normally, standard lookups are created automatically by the system in Dynamics 365 for Finance and Operations and are based on the extended data types and table setup. It is also possible to override the standard functionality by creating your own lookups from the code or using the Dynamics 365 for Finance and Operations forms.

In this chapter, we will cover various lookup types, such as file selector, color picker, or tree lookup, as well as the different approaches to create them.

Creating an automatic lookup

EDT (Extended Data Type) and table relation type lookups are the simplest lookups in Dynamics 365 for Finance and Operations and can be created in seconds without any programming knowledge. They are based on table relations and appear automatically. No additional modifications are required.

This recipe will show you how to create a very basic automatic lookup using table relations. To demonstrate this, we will add a new `Method of payment` column to the existing **Customer group** form.

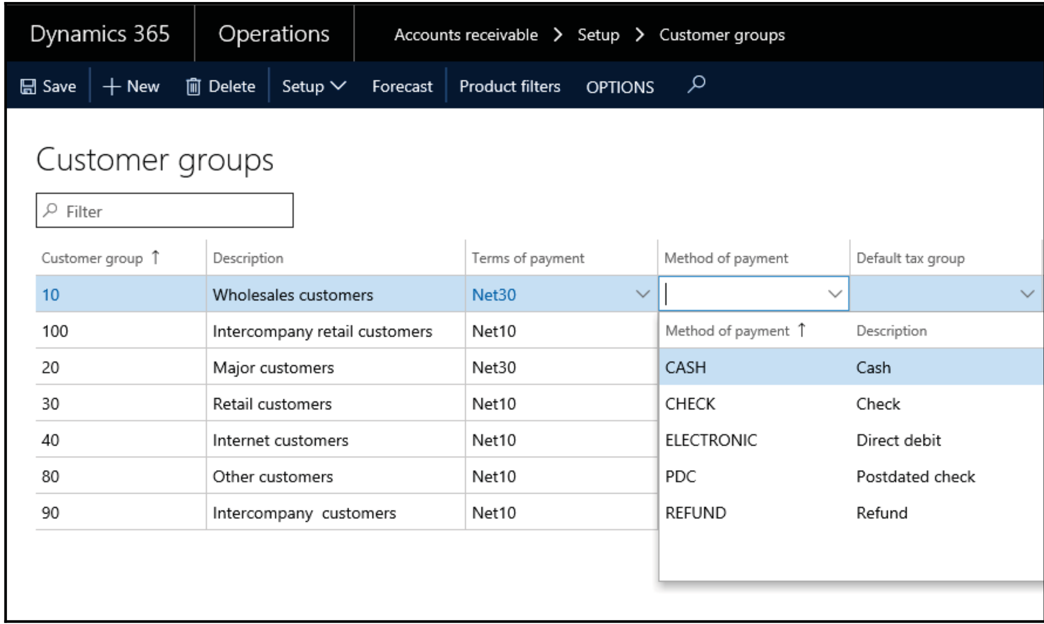
How to do it...

To create an automatic lookup, we can follow the following steps:

1. Create a new solution named `CreatingAutomaticLookup` and assign an appropriate package to it.
2. Find a `CustGroup` table in AOT under application explorer, right-click the table, and select the option `CreateExtension` to add it to the project and create a new `string` type field in the table with the following properties:

Property	Value
Name	PaymMode
ExtendedDataType	CustPaymMode

3. Add the newly created field to the end of the `Overview` field group of the table.
4. Save your object(s) and build the solution.
5. To check the results, navigate to **Accounts receivable** | **Setup** | **Customers** | **Customer groups** and note the newly created **Method of payment** column with the lookup:



How it works...

The newly created `PaymMode` field is based on the `CustPaymMode` extended data type and therefore, it automatically inherits its relation. To follow the best practices, all relations must be present on tables. We also add the newly created field to the table's `Overview` group to make sure that the field automatically appears on the **Customer group** form. This relation ensures that the field has an automatic lookup.

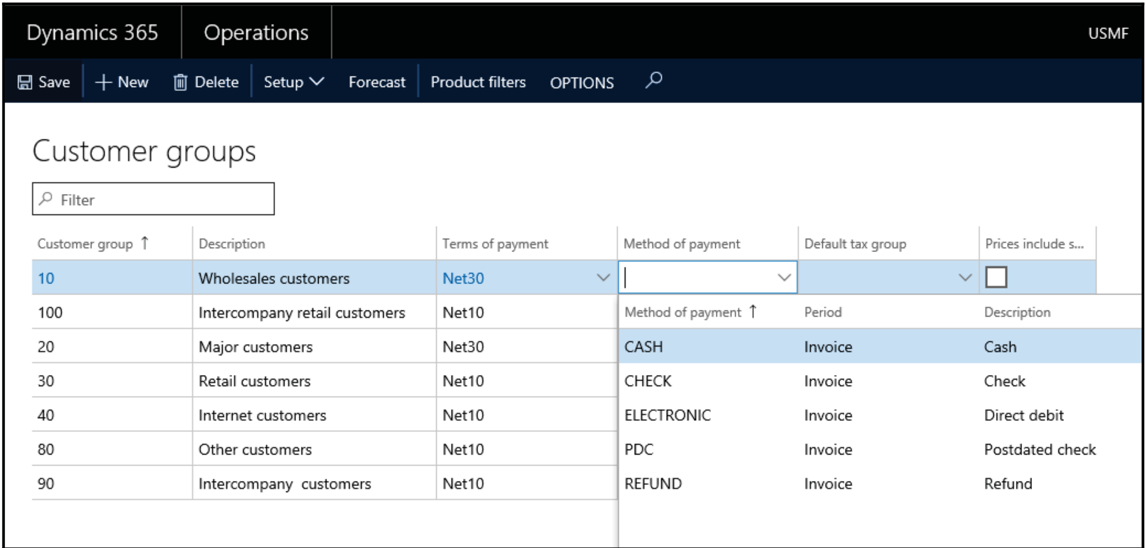
There's more...

The automatically generated lookup, in the preceding example, has only two columns--`Method of payment` and `Description`. Dynamics 365 for Finance and Operations allows us to add more columns or change the existing columns with minimum effort by changing various properties. Lookup columns can be controlled at several different places:

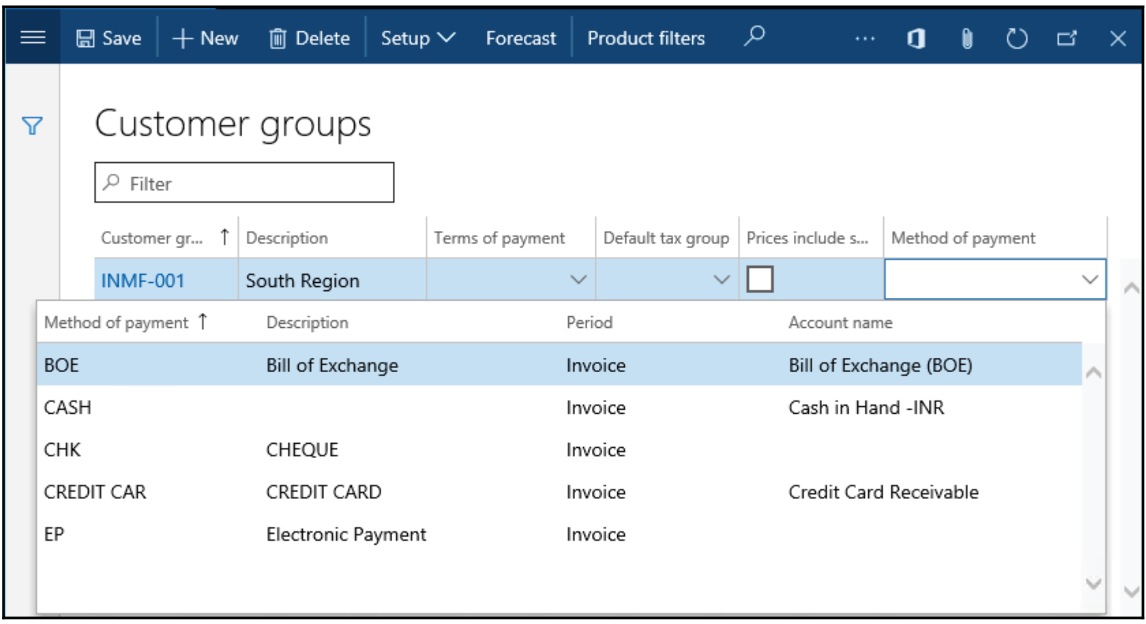
- Relation fields, on either an extended data type or a table, are always shown on lookups as columns.

- Fields defined in the table's `TitleField1` and `TitleField2` properties are also displayed as lookup columns.
- The first field of every table's index is displayed as a column.
- The index fields and the `TitleField1` and `TitleField2` properties are in effect only when the `AutoLookup` field group of a table is empty. Otherwise, the fields defined in the `AutoLookup` group are displayed as lookup columns along with the relation columns.
- Duplicate columns are shown only once.

Now, to demonstrate how the `AutoLookup` group can affect lookup columns, let us modify the previous example by adding an additional field to this group. Let us customize and add the `PaymSumBy` field to the `AutoLookup` group on the `CustPaymModeTable` table in the middle, between the `PaymMode` and `Name` fields. Now, the lookup has one more column labeled **Period**:



It is also possible to add display methods to the lookup's column list. We can extend our example by adding the `paymAccountName()` display method to the `AutoLookup` group on the `CustPaymModeTable` table right after `PaymSumBy`. Save your object and build the project. Check the result now:



Now, in this lookup, we can see the **Account Name** next to **Period**, but with the display method in this lookup we don't have options to filter the records. Methods have limitations to display only records, similar to earlier versions, of Dynamics 365 for Finance and Operations.

Creating a lookup dynamically

Automatic lookups, mentioned in the previous recipe, are widely used across the system and are very useful in simple scenarios. When it comes to showing different fields from different data sources, applying various static or dynamic filters, or similar, some coding is required. The current version of Dynamics 365 for Finance and Operations is flexible enough that the developer can create custom lookups, either using the Dynamics 365 for Finance and Operations forms or by running them dynamically from the **X++ code**.

This recipe will show how to dynamically build a runtime lookup from the code. In this demonstration, we will modify the **Vendor account** lookup on the **Customers** form to allow users to select only those vendors that use the same currency as the currently selected customer.

How to do it...

To create a lookup dynamically, we can follow the following steps:-

1. Create a new solution name `CreatingDynamicLookup` and assign an appropriate package to it.
2. Create a new extension class for the `VendTable` table and add it to the project. Use the following code in the class:

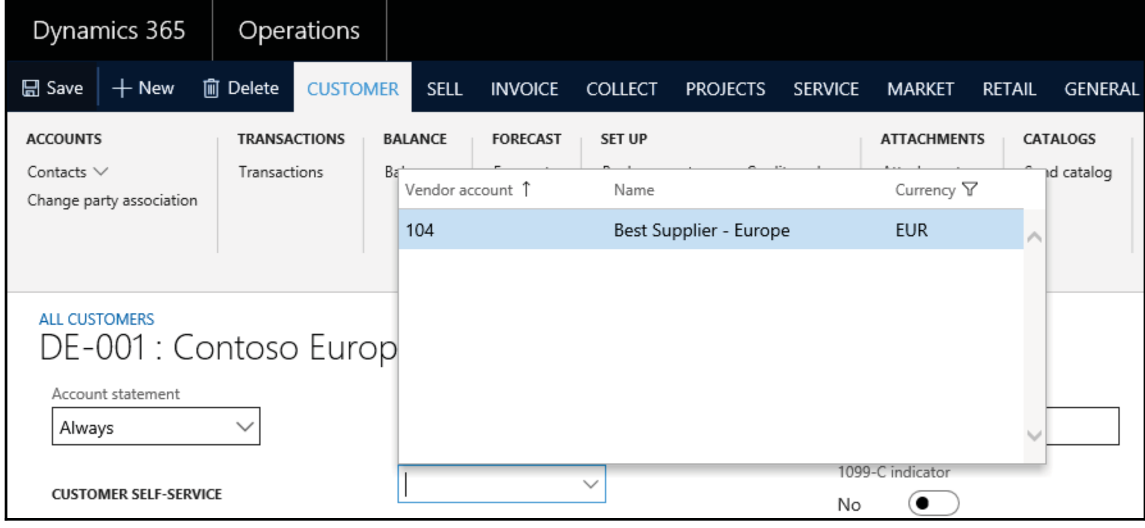
```
[ExtensionOf(tableStr(VendTable))]
final class VendTable_Extension
{
   public static void lookupVendorByCurrency(
   FormControl  _callingControl,
   CurrencyCode _currency)
  {
     Query                   query;
     QueryBuildDataSource    qbds;
     QueryBuildRange         qbr;
     SysTableLookup          lookup;
     query = new Query();
     qbds = query.addDataSource(tableNum(VendTable));
     qbr = qbds.addRange(fieldNum(VendTable,Currency));
     qbr.value(queryvalue(_currency));
     lookup = SysTableLookup::newParameters(
     tableNum(VendTable),
      _callingControl,
      true);
     lookup.parmQuery(query);
     lookup.addLookupField(
     fieldNum(VendTable, AccountNum),
     true);
     lookup.addLookupField(fieldNum(VendTable,Party));
     lookup.addLookupField(fieldNum(VendTable,Currency));
     lookup.performFormLookup();
  }
}
```

3. Now create a new extension class for the `CustTable` form with the name `CustTable_Extension`. Find form `CustTable` and then, in the design, locate the `VendAccount` field at `CustTable\Design\Tab\TabPageDetails\TabHeader\TabDetails\Vendor\Vendor_VendAccount`, copy its `OnLookup()` event, and paste it in an extension class with the following code snippet:

```
[ExtensionOf(formStr(CustTable))]
```

```
final class CustTable_Extension
{
  [FormControlEventHandler(formControlStr(CustTable,
  Vendor_VendAccount), FormControlEventType::Lookup)]
  public void Vendor_VendAccount_OnLookup(
  FormControl sender,FormControlEventArgs e)
  {
    VendTable::lookupVendorByCurrency(
    sender, this.CustTable.Currency);
    FormControlCancelableSuperEventArgs cancelSuper =
    e as FormControlCancelableSuperEventArgs;

    //cancel super() to prevent error.
    cancelSuper.CancelSuperCall();
  }
}
```

4. To test this, navigate to **Accounts receivable** | **Common** | **Customers** | **All customers**, select any of the customers, and click on **Edit** in the action pane. Once the **Customers** form is displayed, expand the **Vendor account** lookup located in the **Miscellaneous details** tab page, under the **Remittance** group. The modified lookup now has an additional column named **Currency**, and vendors in the list will match the customer's currency. The following screenshot depicts this:



How it works...

First, on the `VendTable` table, we create a new method that generates the lookup. This is the most convenient place for such a method, taking into consideration that it may be reused at a number of other places.

In this method, we first create a new query, which will be the base for lookup records. In this query, we add a new data source based on the `VendTable` table and define a new range based on the `Currency` field.

Next, we create the actual lookup object and pass the query object through the `parmQuery()` member method. The `lookup` object is created using the `newParameters()` constructor of the `SysTableLookup` class. It accepts the following three parameters:

- The table ID, which is going to be displayed.
- A reference to the form calling the control.
- An optional `boolean` value, which specifies that the value in the control should be preselected in the lookup. The default is `true`.

We use the `addLookupField()` method to add three columns--`Vendor account`, `Name`, and `Currency`. This method accepts the following parameters:

- The ID of the field that will be displayed as a column.
- An optional Boolean parameter that defines which column will be used as a return value to the caller control upon user selection. Only one column can be marked as a return value. In our case, it is vendor account.

Finally, we run the lookup by calling the `performFormLookup()` method.

The last thing to do is to add some code to the `lookup()` method of the **VendAccount** field of the **CustTable** data source in the **CustTable** form. By replacing its `super()` method with our custom code, we have overridden the standard, automatically generated lookup, with the custom lookup.

There's more...

Suppose for some use case we need to show the vendor balance as well. We could achieve this by using the method `balanceAllCurrency` on `VendTable`, which displays the vendor balance. To do so, we could add the following code in the `lookupVendorByCurrency` method in the `VendTable_Extension` class before the `performFormLookup()` method call:

```
lookup.addLookupField(fieldNum(VendTable,Party));
lookup.addLookupField(fieldNum(VendTable,Currency));
lookup.addLookupMethod
 (tableMethodStr(VendTable,balanceAllCurrency));
lookup.performFormLookup();
```

Using a form to build a lookup

For the most complex scenarios, where you need some advance lookups with more options on data filtration or selection; Dynamics 365 for Finance and Operations offers the possibility to create and use a form as a lookup, by creating a new form.

Similar to forms, the form lookups support various features, such as tab pages, event handling, complex logic, and so on. In this recipe, we will demonstrate how to create a lookup using a form. As an example, we will modify the standard customer account lookup to display only the customers who are not placed on hold for invoicing and delivery.

How to do it...

1. Add a new project and create a new form named `CustLookup`. Add a new data source with the following properties:

Property	Value
Name	CustTable
Table	CustTable
AllowCheck	No
AllowEdit	No
AllowCreate	No

AllowDelete	No
OnlyFetchActive	Yes

2. Right-click on **Design** and **Add form Pattern** `Lookup - Basic` on design.
3. Add a new grid control to the form's design with the following properties:

Property	**Value**
Name	Customers
ShowRowLabels	No
DataSource	CustTable

4. Drag and drop `AccountNum` and `Blocked` fields from a `CustTable` data source. Set property `auto declaration` to `Yes`.
5. Add a new `ReferenceGroup` control to the grid with the following properties, right after `AccountNum`:

Property	**Value**
Name	Name
DataSource	CustTable
ReferenceField	Party

6. Add one more `StringEdit` control to the grid with the following properties, right after the `Name`:

Property	**Value**
Name	Phone
DataSource	CustTable
DataMethod	phone

7. Override the form's `init()` method with the following code snippet:

```
public void init()
{
   super();
   element.selectMode(CustTable_AccountNum);
}
```

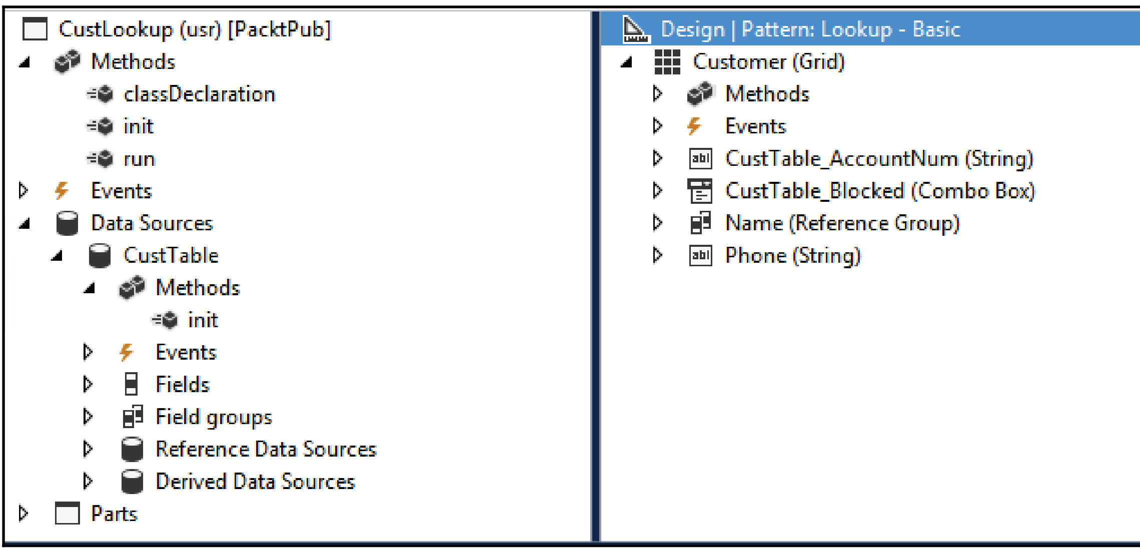
8. Override the form's `run()` method with the following code snippet:

```
public void run()
{
   FormStringControl callingControl;
   boolean           filterLookup;
   callingControl = SysTableLookup::getCallerStringControl(
   element.args());
   filterLookup = SysTableLookup::filterLookupPreRun(
   callingControl,
   CustTable_AccountNum,
   CustTable_ds);
   super();
   SysTableLookup::filterLookupPostRun(
   filterLookup,
   callingControl.text(),
   CustTable_AccountNum,
   CustTable_ds);
}
```

9. Finally, override the `init()` method of the `CustTable` data source with the following code snippet:

```
public void init()
{
   Query                 query;
   QueryBuildDataSource qbds;
   QueryBuildRange      qbr;
   query = new Query();
   qbds  = query.addDataSource(tableNum(CustTable));
   qbr = qbds.addRange(fieldNum(CustTable,Blocked));
   qbr.value(queryvalue(CustVendorBlocked::No));
   this.query(query);
}
```

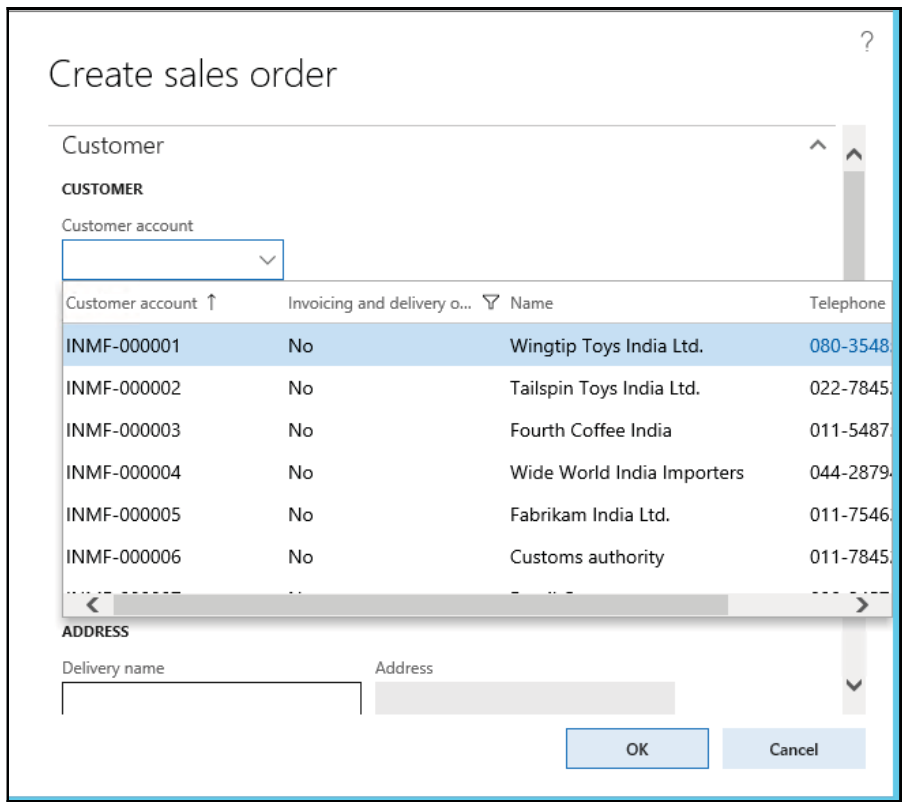
10. The form in the **Application Object Tree** (**AOT**) will look similar to the following screenshot:



11. Locate the `CustAccount` extended data type in the AOT, create its extension with name `CustAccount.Extension`, and change its property as follows:

Property	Value
FormHelp	CustLookup

12. To test the results, navigate to **Sales and marketing** | **Common** | **Sales orders** | **All sales orders** and start creating a new sales order. Note that, now, the **Customer account** lookup is different, and it includes only active customers:



How it works...

Automatically generated lookups have a limited set of features and are not suitable in more complex scenarios. In this recipe, we are creating a brand new form-based lookup, which will replace the existing customer account lookup. The name of the newly created form is `CustLookup` and it contains the `Lookup` text at the end to make sure it can be easily distinguished from other forms in the AOT.

In the form, we add a new data source and change its properties. We do not allow any data updating by setting the `AllowEdit`, `AllowCreate`, and `AllowDelete` properties to `No`. Security checks will be disabled by setting `AllowCheck` to `No`. To increase the performance, we set `OnlyFetchActive` to `Yes`, which will reduce the size of the database result set to only the fields that are visible on the form. We also set the data source index to define initial data sorting.

Next, in order to make our form lookup look exactly like a standard lookup, we have to adjust its layout. Therefore, we set its `Frame` and `WindowType` properties to `Border` and `Popup`, respectively. This removes form borders and makes the form very similar to a standard lookup. Then, we add a new grid control with four controls inside, which are bound to the relevant `CustTable` table fields and methods. We set the `ShowRowLabels` property of the grid to `No` to hide the grid's row labels.

After this, we have to define which form control will be used to return a value from the lookup to the calling form. We need to specify the form control manually in the form's `init()` method, by calling `element.selectMode()`, with the name of the control as an argument.

In the form's `run()` method, we add some filtering, which allows the user to use the `asterisk` (`*`) symbol to search for records in the lookup. For example, if the user types `1*` into the `Customer account` control, the lookup will open automatically with all customer accounts starting with 1. To achieve this, we use the `filterLookupPreRun()` and `filterLookupPostRun()` methods of the standard `SysTableLookup` class. Both these methods require a calling control, which can be obtained by the `getCallerStringControl()` method of the same `SysTableLookup` class. The first method reads the user input and returns `true` if a search is being performed, otherwise, it returns `false`. It must be called before the `super()` method in the form's `run()` method, and it accepts four arguments:

- The `calling` control on the parent form
- The `returning` control on the lookup form
- The `main data source` on the lookup form
- An `optional list` of other data sources on the lookup form, which are used in the search

The `filterLookupPostRun()` method must be called after the `super()` method in the form's `run()` method, and it also accepts four arguments:

- A result value from the previously called `filterLookupPreRun()` method
- The user text specified in the calling control
- The returning control on the lookup form
- The lookup data source

The code in the `CustTable` data source's `init()` method replaces the data source query created by its `super()` method with the custom one. Basically, here, we create a new `Query` object and change its range to include only active customers.

The `FormHelp` property of the `CustAccount` extended data type will make sure that this form is opened every time the user opens the **Customer account** lookup.

See also

- The *Building a query object* recipe in `Chapter 1`, *Processing Data*

Building a tree lookup

The form's `tree` controls are a user-friendly way of displaying a hierarchy of related records, such as a company's organizational structure, inventory bill of materials, projects with their subprojects, and so on. These hierarchies can also be displayed in the custom lookups, allowing users to browse and select the required value in a more convenient way.

The *Using a tree control* recipe in `Chapter 2`, *Working with Forms*, explained how to present the budget model hierarchy as a tree in the **Budget model** form. In this recipe, we will reuse the previously created `BudgetModelTree` class and demonstrate how to build a *budget model tree lookup***.**

How to do it...

1. In the AOT, create a new form named `BudgetModelLookup`. Set its design properties as follows:

Property	Value
Frame	Border
WindowType	Popup

2. Add a new `Tree` control to the design with the following properties:

Property	Value
Name	ModelTree
Width	250

3. Add the following line of code to the form's class declaration:

```
BudgetModelTree budgetModelTree;
```

4. Override the form's `init()` method with the following code snippet:

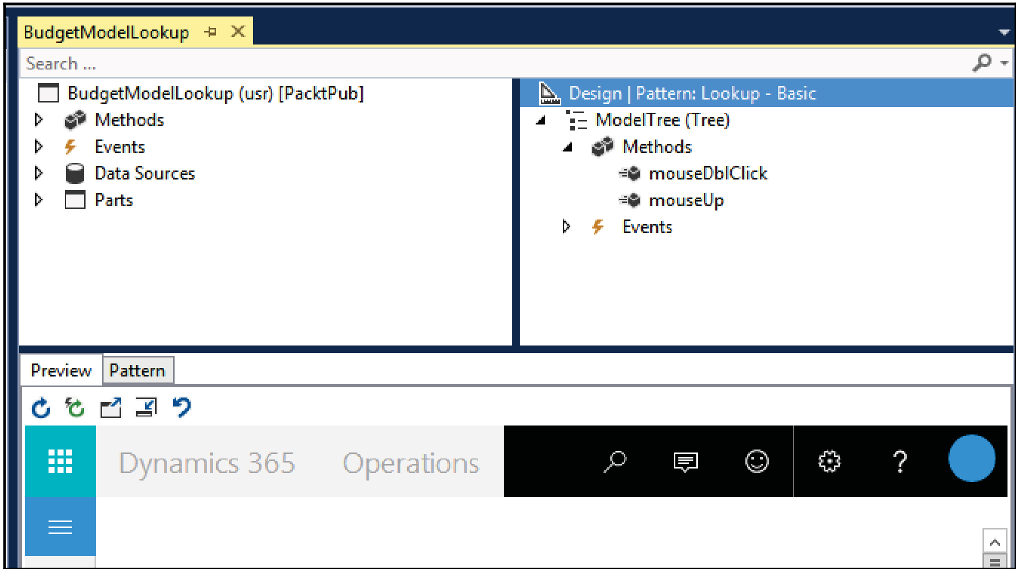
```
 public void init()
{
   FormStringControl callingControl;
   callingControl = SysTableLookup::getCallerStringControl(
   this.args());
   super();
   budgetModelTree = BudgetModelTree::construct(
   ModelTree,
   callingControl.text());
   budgetModelTree.buildTree();
}
```

5. Override the `mouseDblClick()` and `mouseUp()` methods of the `ModelTree` control with the following code snippet:

```
public int mouseDblClick(
int _x,
int _y,
int _button,
boolean _ctrl,
boolean _shift)
{
   int ret;
   FormTreeItem formTreeItem;
   BudgetModel  budgetModel;
   ret = super(_x, _y, _button, _ctrl, _shift);
   formTreeItem = this.getItem(this.getSelection());
   select firstOnly SubModelId from budgetModel
   where budgetModel.RecId == formTreeItem.data();
   element.closeSelect(budgetModel.SubModelId);
   return ret;
}
public int mouseUp(
 int _x,
 int _y,
 int _button,
 boolean _ctrl,
 boolean _shift)
 {
    int ret;
    ret = super(_x, _y, _button, _ctrl, _shift);
```

```
        return 1;
    }
```

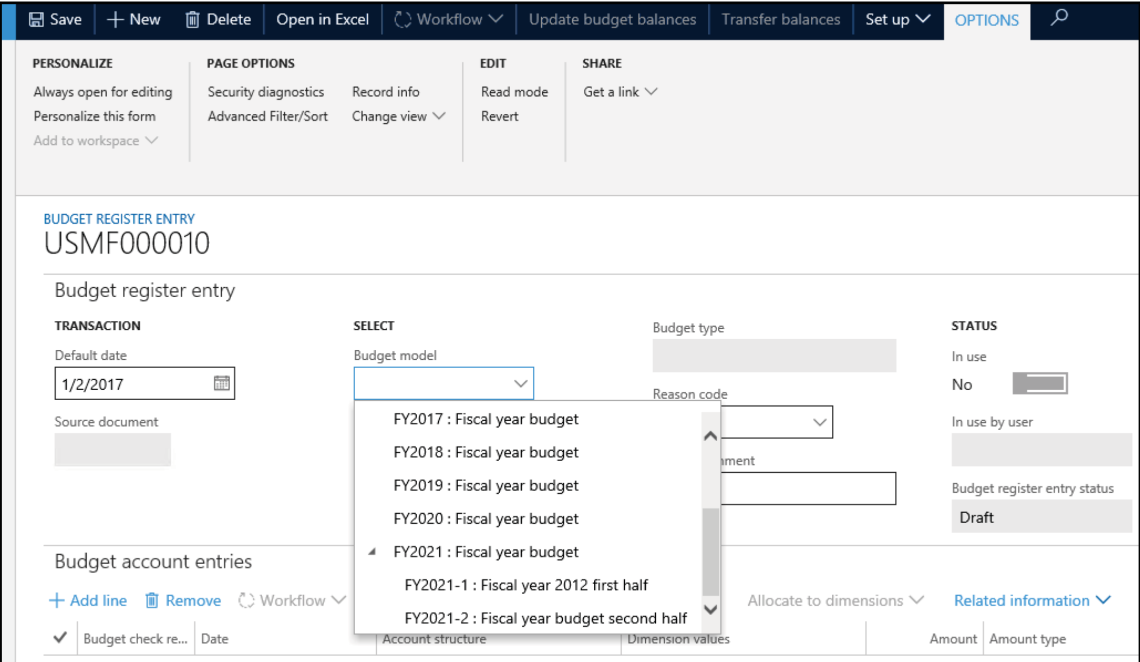
6. The form will look similar to the following screenshot:



7. In the AOT, open the `BudgetModel` table and change its `lookupBudgetModel()` method with the following code snippet:

```
  public static void lookupBudgetModel(
  FormStringControl _ctrl,
  boolean _showStopped = false)
{
    Args     args;
    Object   formRun;
    args = new Args();
    args.name(formStr(BudgetModelLookup));
    args.caller(_ctrl);
    formRun = classfactory.formRunClass(args);
    formRun.init();
    _ctrl.performFormLookup(formRun);
}
```

8. To see the results, navigate to **Budgeting** | **Common** | **Budget register entries** | **All budget register entries**. Start creating a new entry by clicking on the **Budget register entry** button in the action pane and expanding the **Budget model** lookup:



How it works...

First, we create a new form named **BudgetModelLookup**, which we will use as a custom lookup. We set its design's `Frame` and `WindowType` to `Border` and `Popup` respectively, to change the layout of the form so that it looks like a lookup. We also add a new `Tree` control and set its width.

In the form's class declaration, we define the `BudgetModelTree` class, which we have already created in the *Using tree controls* recipe in `Chapter 2`, *Working with Forms*.

The code in the form's `init()` method builds the tree. Here, we create a new object of the `BudgetModelTree` type by calling the `construct()` constructor, which accepts two arguments:

- The `Tree` control, which represents the actual tree.
- The `Budget` model, which is going to be preselected initially. Normally, it is a value of the calling control, which can be detected using the `getCallerStringControl()` method of the `SysTableLookup` application class.
- The code in `mouseDblClick()` returns the user-selected value from the tree node back to the calling control and closes the lookup.
- Finally, the `mouseUp()` method has to be overridden to return 1 to make sure that the lookup does not close while the user expands or collapses the tree nodes.

See also

- The *Using a tree control* recipe in `Chapter 2`, *Working with Forms*

Displaying a list of custom options

Besides normal lookups, Dynamics 365 for Finance and Operations provides a number of other ways to present the available data for user selection. It does not necessarily have to be a record from the database; it can be a list of **hardcoded** options or some external data. Normally, such lists are much smaller as opposed to those of the data-driven lookups, and are used for very specific tasks.

In this recipe, we will create a lookup of several predefined options. We will use a job for this demonstration.

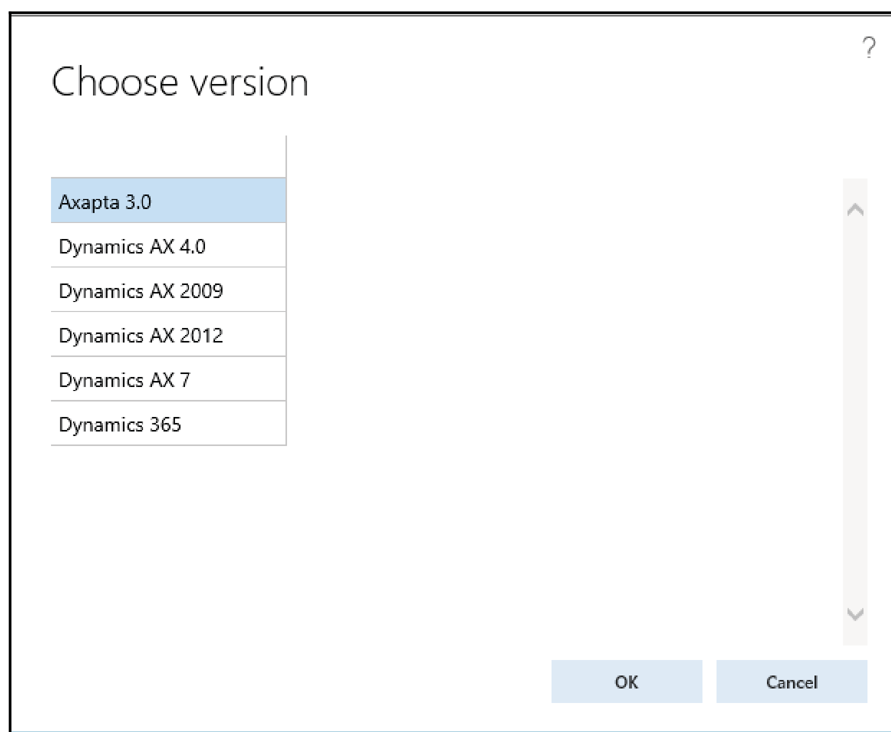
How to do it...

1. Add a new project and add a new `RunnableClass` named `PickList`:

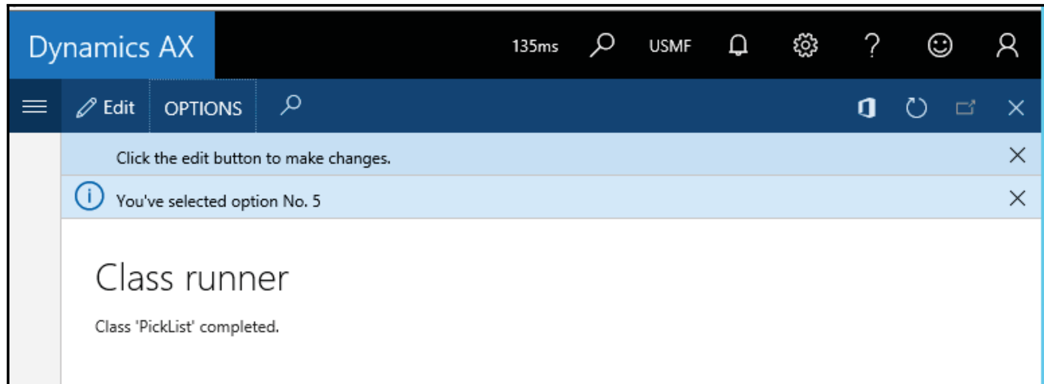
```
class PickList
{
    /// <summary>
    /// Runs the class with the specified arguments.
    /// </summary>
    /// <param name = "_args">The specified arguments.</param>
    public static void main(Args _args)
    {
        Map choices;
        str ret;
        choices = new Map(
        Types::Integer,
        Types::String);
        choices.insert(1, "Axapta 3.0");
        choices.insert(2, "Dynamics AX 4.0");
        choices.insert(3, "Dynamics AX 2009");
        choices.insert(4, "Dynamics AX 2012");
        choices.insert(5, "Dynamics AX 7");
        choices.insert(6, "Dynamics 365");

        ret = pickList(choices, "", "Choose version");
        if (ret)
        {
            info(strFmt("You've selected option No. %1", ret));
        }
    }
}
```

2. Save all your code, right-click on this new class, and click on `Set as startup object`. Now build your project.

3. Run the project to view the results:



4. Double-click on one of the options to show the selected option in the **Infolog** window:



How it works...

The key element in this recipe is the global `pickList()` function. Lookups created using this function are based on values stored in a map. In our example, we define and initialize a new map. Then, we insert a few key-value pairs and pass the map to the `pickList()` function. This function accepts three parameters:

- A **map** that contains lookup values
- A **column header**, which is not used here
- A **lookup title**

The function that displays values from the map returns the corresponding keys, once the option is selected.

There's more...

The global `pickList()` function can basically display any list of values. Besides that, Dynamics 365 for Finance and Operations also provides a number of other global lookup functions, which can be used in more specific scenarios. Here are a few of them:

- `pickDataArea()`: This shows a list of Dynamics 365 for Finance and Operations companies.
- `pickUserGroups()`: This shows a list of user groups in the system.
- `pickUser()`: This shows a list of Dynamics 365 for Finance and Operations users.
- `pickTable()`: This shows all Dynamics 365 for Finance and Operations tables.
- `pickField()`: This shows table fields. The table number has to be specified as an argument for the function.
- `pickClass()`: This shows a list of Dynamics 365 for Finance and Operations classes.

Displaying custom options in another way

The global system functions, such as `pickList()` and `pickUser()`, allow developers to build various lookups displaying a list of custom options. Besides that, the standard Dynamics 365 for Finance and Operations application contains a few more useful functions, allowing us to build more complex lookups of custom options.

One of the custom functions is called `selectSingle()`, and it provides the user with a list of options. It also displays a checkbox next to each option that allows users to select the option. To demonstrate this, we will create a new class that shows the usage of these functions.

How to do it...

1. Add a new project in the solution name `DisplayCustomOptionsAnotherWay`.
2. Find a `SysListSelect` form in AOT and select customize to add it in the project.
3. Add new method `selectSingle` and add the following code:

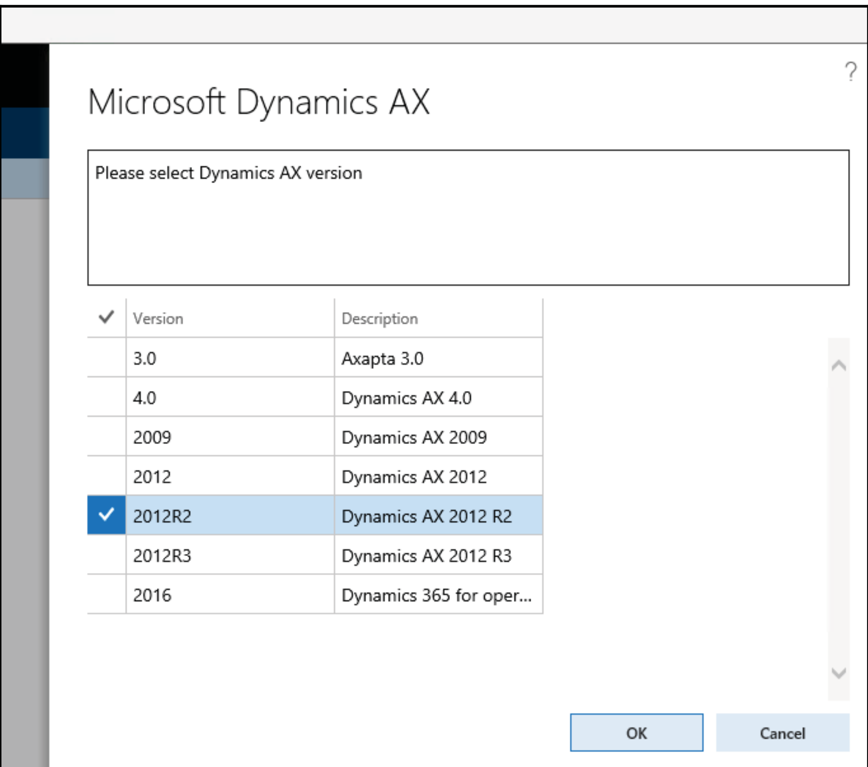
```
public void selectSingle()
    {
       singleSelect = true;
    }
```

4. Now select project and add a new item, a new Runnable class (Job) named `SysListSelectSingle`:

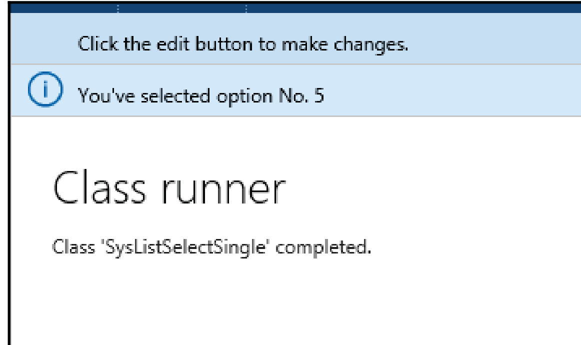
```
class SysListSelectSingle
{
   /// <summary>
   /// Runs the class with the specified arguments.
   /// </summary>
   /// <param name = "_args">The specified arguments.</param>
   public static void main(Args _args)
  {
     container choices;
     container headers;
     container selection;
     container selected;
     boolean   ok;
     choices = [
     ["3.0\nAxapta 3.0", 1, false],
     ["4.0\nDynamics AX 4.0", 2, false],
     ["2009\nDynamics AX 2009", 3, false],
     ["2012\nDynamics AX 2012", 4, false],
     ["2012R2\nDynamics AX 2012 R2", 5, false],
     ["2012R3\nDynamics AX 2012 R3", 6, true],
     ["2016\nDynamics 365 for operations", 7, true]];
     headers = ["Version", "Description"];
     selection = SysListSelectSingle::selectSingle(
     "Choose version",
     "Please select Dynamics AX version",
```

```
            choices,
            headers);
            [ok, selected] = selection;
            if (ok && conLen(selected))
            {
               info(strFmt(
               "You've selected option No. %1",
                conPeek(selected,1)));
            }
         }
            static client container selectSingle(
            Caption     _caption,
            str  _info,// An info text displayed in the top of the form
            container _choices,
            container _headers = conNull(),// If null, the list view is
            used
            Object      _caller = null
            )
           {
             Args                    args;
             FormRun                 formRun;
             Object                  obj;
             container               selected;
             args = new Args(formStr(SysListSelect));
             args.caller(_caller);
             formRun = classfactory.formRunClass(args);
             formRun.init();
             formRun.design().visible(true);
             obj = formRun;
             obj.infotxt(_info);
             obj.choices(_choices);
             obj.headers(_headers);
             obj.selectSingle();
             formRun.run();
             formRun.wait();
             selected = obj.selected();
             if (conLen(selected) > 0)
            {
               return [formRun.closedOk(),[conPeek(selected,1)]];
            }
             return [formRun.closedOk(), conNull()];
           }
```

5. Run the job to display the options:



6. Select any of the options, click on the **OK** button, and note that your choice is displayed in the **Infolog** window shown in the following screenshot:



How it works...

We start by defining the choices variable and setting its values. The variable is a container and it holds container values, where each container inside the parent container is made of three elements and represents one selectable option in the list:

- The first element is text displayed on the lookup. By default, in the lookup, only one column is displayed, but it is possible to define more columns, simply by separating the texts using the **new line symbol**.
- The second element is the number of an item in the list. This value is returned from the lookup.
- The third value specifies whether the option is marked by default.

Now, when the list values are ready, we call the `selectSingle()` function to build the actual lookup. This function accepts five arguments:

- The **window** title
- The lookup **description**
- A **containe**r of list values
- A container representing **column headings**
- An **optional reference** to a caller object

The `singleSelect()` function returns a container of two elements:

- `true` or `false` depending whether the lookup was closed using the **OK** button or not
- The numeric value of the selected option

There's more...

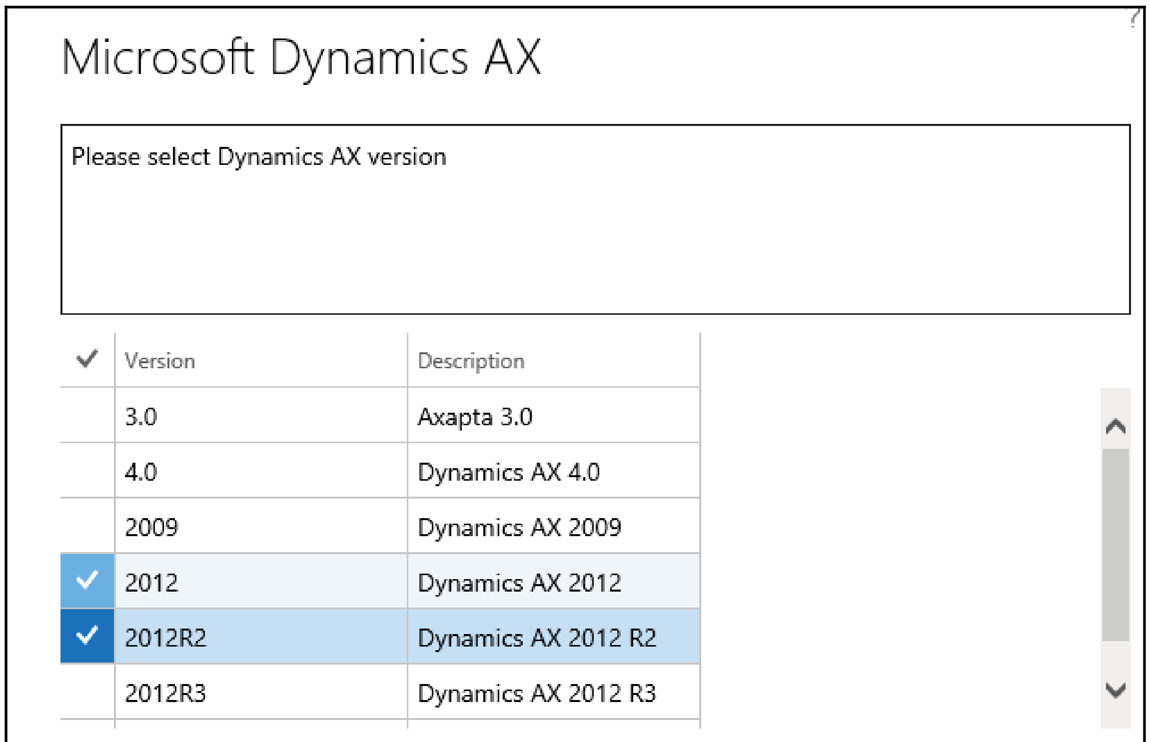
You may notice that the lookup, which was created using the `singleSelect()` method, allows chooses only one option from the list. There is another similar function named `selectMultiple()`, which is exactly the same except that the user can select multiple options from the list. The following code snippet demonstrates its usage:

```
class SysListSelectMultiple
{
    /// <summary>
    /// Runs the class with the specified arguments.
    /// </summary>
    /// <param name = "_args">The specified arguments.</param>
```

```
public static void main(Args _args)
{
   container choices;
   container headers;
   container selection;
   container selected;
   boolean   ok;
   choices = [
   ["3.0\nAxapta 3.0", 1, false],
   ["4.0\nDynamics AX 4.0", 2, false],
   ["2009\nDynamics AX 2009", 3, false],
   ["2012\nDynamics AX 2012", 4, false],
   ["2012R2\nDynamics AX 2012 R2", 5, false],
   ["2012R3\nDynamics AX 2012 R3", 6, true],
   ["2016\nDynamics 365 for operations", 7, true]];
   headers = ["Version", "Description"];
   selection = SysListSelectMultiple::selectMultiple(
   "Choose version",
   "Please select Dynamics AX version",
   choices,
   headers);
   [ok, selected] = selection;
   if (ok && conLen(selected) > 0)
   {
      for (int i = 1; i <= conLen(selected); i++)
         {
            info(strFmt(
            "You've selected option No. %1",
             conPeek(selected,i)));
         }
   }
}
  /* Returns container with the status of how the form is
  closed plus the selected ids.*/
  static client container selectMultiple(
  Caption  _caption,
  str _info, // An info text displayed in the top of the form
  container _choices,
  container _headers = conNull(), // If null, the list view
  is  used
  Object      _caller = null
  )
 {
    Args                    args;
    FormRun                 formRun;
    Object                  obj;
    args = new Args(formStr(SysListSelect));
    args.caller(_caller);
```

```
            formRun = classfactory.formRunClass(args);
            formRun.init();
            formRun.design().visible(true);
            obj = formRun;
            obj.infotxt(_info);
            obj.choices(_choices);
            obj.headers(_headers);
            formRun.run();
            formRun.wait();
            return [formRun.closedOk(),obj.selected()];
        }
    }
```

Now, in the lookup, it is possible to select multiple options:



Note that in this case, the returned value is a container holding the selected options.

Building a lookup based on the record description

Normally, data lookups in Dynamics 365 for Finance and Operations display a list of records where the first column always contains a value, which is returned to a calling form. The first column in the lookup normally contains a **unique record identification value**, which is used to build relations between tables. For example, the **Customer** lookup displays the customer account number, the customer name, and some other fields; the **Inventory item** lookup displays the item number, the item name, and other fields.

In some cases, the record identifier can be not so informative. For example, it is much more convenient to display a person's name versus its number. In the standard application, you can find a number of places where the contact person is displayed as a person's name, even though the actual table relation is based on the contact person's ID.

In this recipe, we will create such a lookup. We will replace the **Vendor group selection** lookup on the **Vendors** form to show group description, instead of group ID.

How to do it...

1. In the AOT, create a new `String` extended data type with the following properties:

Property	Value
Name	VendGroupDescriptionExt
Label	Group
Extends	Description

2. Open the `VendTable` table and create a new method with the following code snippet:

```
public edit VendGroupDescriptionExt editVendGroup(
 boolean                 _set,
 VendGroupDescriptionExt _group)
{
   VendGroup vendGroup;
 if (_set)
 {
   if (_group)
```

```
        {
            if (VendGroup::exist(_group))
            {
                this.VendGroup = _group;
            }
            else
           {
               select firstOnly VendGroup from vendGroup
               where vendGroup.Name == _group;
               this.VendGroup = vendGroup.VendGroup;
           }
        }
               else
               {
                   this.VendGroup = '';
               }
      }
        return VendGroup::name(this.VendGroup);
    }
```

3. In the AOT, find the `VendTable` form, locate the `Posting group` control inside **MainTab** | **TabPageDetails** | **Tab** | **TabGeneral** | **UpperGroup** | **Posting** , and modify its properties as follows:

Property	**Value**
DataGroup	

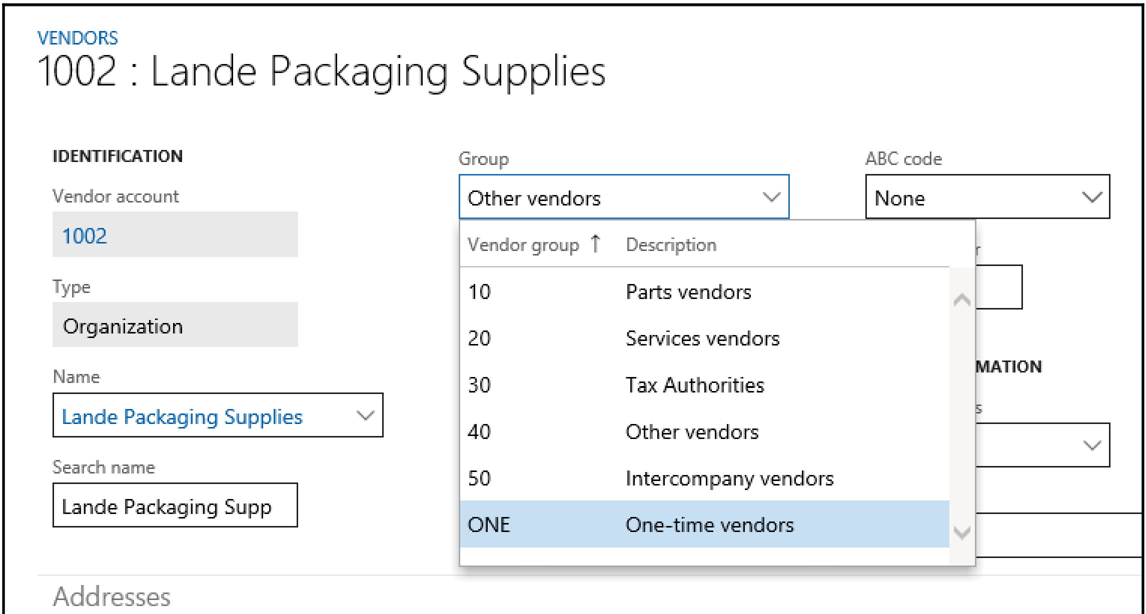
4. In the same form, in the `Posting` group, modify the `Posting_VendGroup` control as follows:

Property	Value
DataField	
DataMethod	editVendGroup

5. Override the `lookup()` method of the `Posting_VendGroup` control with the following code snippet:

```
public void lookup()
{
    this.performTypeLookup(extendedTypeNum(VendGroupId));
}
```

6. To check the results, navigate to **Accounts payable** | **Common** | **Vendors** | **All vendors**, select any record, and click on the **Edit** button in the action pane. In the opened form, check the newly created lookup on the `Group` control, located in the **General** tab of the page:



How it works...

First, we create a new extended data type, which we will use as the basis for the `Vendor group selection` control. The type extends the existing `Description` extended data type as it has to be of the same size as the vendor group name. It will also have the same label as `VendGroupId` because it is going to replace the existing `Group` control on the form.

Next, we create a new `edit` method, which is used to show the **group description** instead of the **group ID** on the form. It also allows changing the control value.

The `edit` method is created on the `VendTable` table, it is the most convenient place for reuse and it uses the newly created extended data type. This ensures that the label of the `user` control stays the same. The method accepts two arguments, as this is a mandatory requirement for the `edit` methods. The first argument defines whether the control was modified by the user, and if yes, the second argument holds the modified value. In this recipe, the second value can be either `group ID` or `group description`. The value will be group ID if the user selects this value from the lookup. It will be group description if the user decides to manually type the value into the control. We use the extended data type, which is bigger in size, that is, the `VendGroupDescriptionExt` type. The method returns a vendor group name, which is shown on the form.

Next, we need to modify the `VendTable` form. We change the existing vendor group ID control to use the newly created `edit` method. By doing this, we make the control unbound and therefore lose the standard lookup functionality. To correct this, we override the `lookup()` method on the control. Here, we use the `performTypeLookup()` method to restore the lookup functionality.

There's more...

In the previous example, you may notice that the lookup does not find the currently selected group. This is because the system tries to search group ID by group description. This section will show how to solve this issue.

First, we have to create a new form named **VendGroupLookup**, that acts as a lookup. Add a new data source to the form, with the following properties:

Property	**Value**
Name	VendGroup
Table	VendGroup
Index	GroupIdx
AllowCheck	No
AllowEdit	No
AllowCreate	No
AllowDelete	No
OnlyFetchActive	Yes

Change the properties of the form's design as follows:

Property	Value
Frame	Border
WindowType	Popup

Add a new `Grid` control to the form's design with the following properties:

Property	Value
Name	VendGroups
ShowRowLabels	No
DataSource	VendGroup
DataGroup	Overview

Several new controls will appear in the grid automatically. Change the properties of the `VendGroups_VendGroup` control as follows:

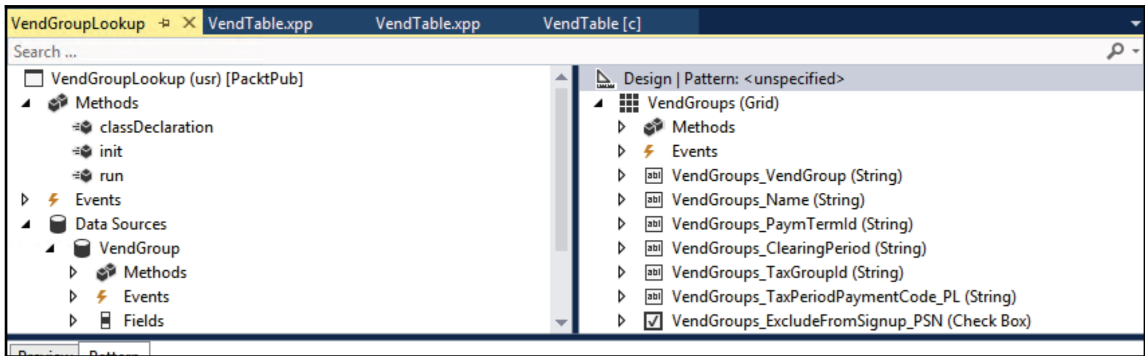
Property	Value
AutoDeclaration	Yes

Override the form's `init()` and `run()` methods with the following code snippet, respectively:

```
public void init()
{
   super();
   element.selectMode(VendGroups_VendGroup);
}
public void run()
{
   VendGroupId groupId;
   groupId = element.args().lookupValue();
   super();
   VendGroup_ds.findValue(
   fieldNum(VendGroup,VendGroup), groupId);
}
```

The key element here is the `findValue()` method in the form's `run()` method. It places the cursor on the currently selected vendor group record. The group ID is retrieved from the argument's object using the `lookupValue()` method.

In the project, the form design will look similar to the following screenshot:



Next, we need to create a new `static` method on the `VendGroup` table, which opens the new lookup form:

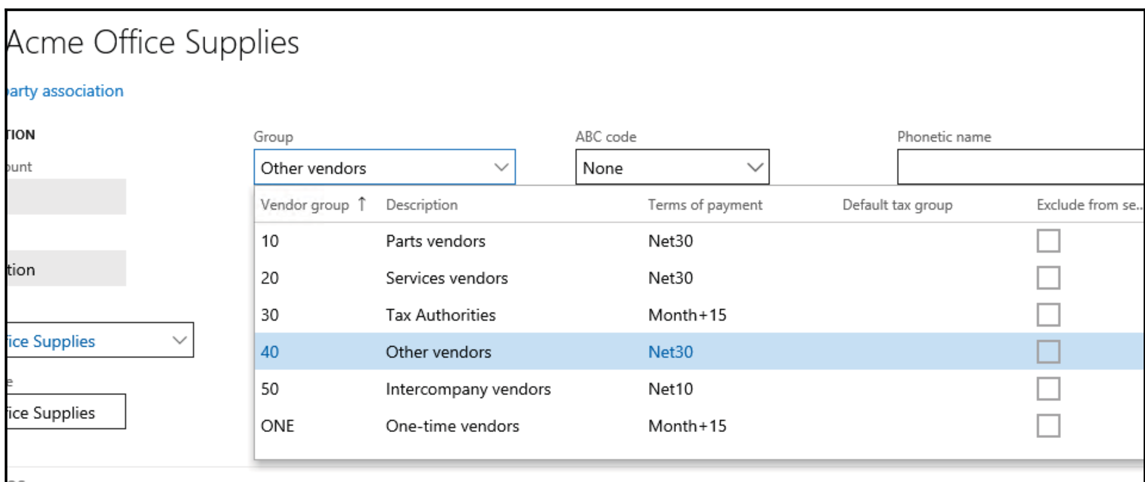
```
    public static void lookupVendorGroupForm(
    FormStringControl _callingControl,
    VendGroupId       _groupId)
{
    FormRun formRun;
    Args    args;
    args = new Args();
    args.name(formStr(VendGroupLookup));
    args.lookupValue(_groupId);
    formRun = classFactory.formRunClass(args);
    formRun.init();
    _callingControl.performFormLookup(formRun);
}
```

Here, we use the `formRunClass()` method of the global `classFactory` object. Note that here we pass the group ID to the form through the `Args` object.

The final touch is to change the code in the `lookup()` method of the `VendGroups_VendGroup` control on the `VendTable` form:

```
public void lookup()
{
   VendGroup::lookupVendorGroupForm(this, VendTable.VendGroup);
}
```

Now, when you open the **Vendors** form, make sure that the current vendor group in the **Group** lookup is preselected correctly:



Building the browse for folder lookup

In Dynamics 365 for Finance and Operations, file reading or saving is a very common operation. Normally, for non-automated operations, the system prompts the user for file input.

This recipe will demonstrate how the user can be presented with the file browse dialog box in order to choose the files in a convenient way.

Folder browsing lookups can be used when the user is required to specify a local or a network folder, to store or retrieve external files. Such lookups are generated in Dynamics 365 for Finance and Operations using the `File upload` control.

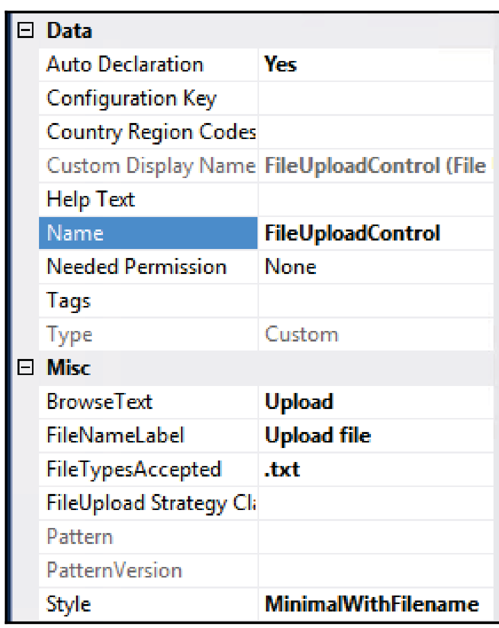
In this recipe, we will learn how to create a lookup for folder browsing. As an example, we will create a new field and control named `Documents` on the **Vendor parameters** form, which will allow us to store a folder path.

How to do it...

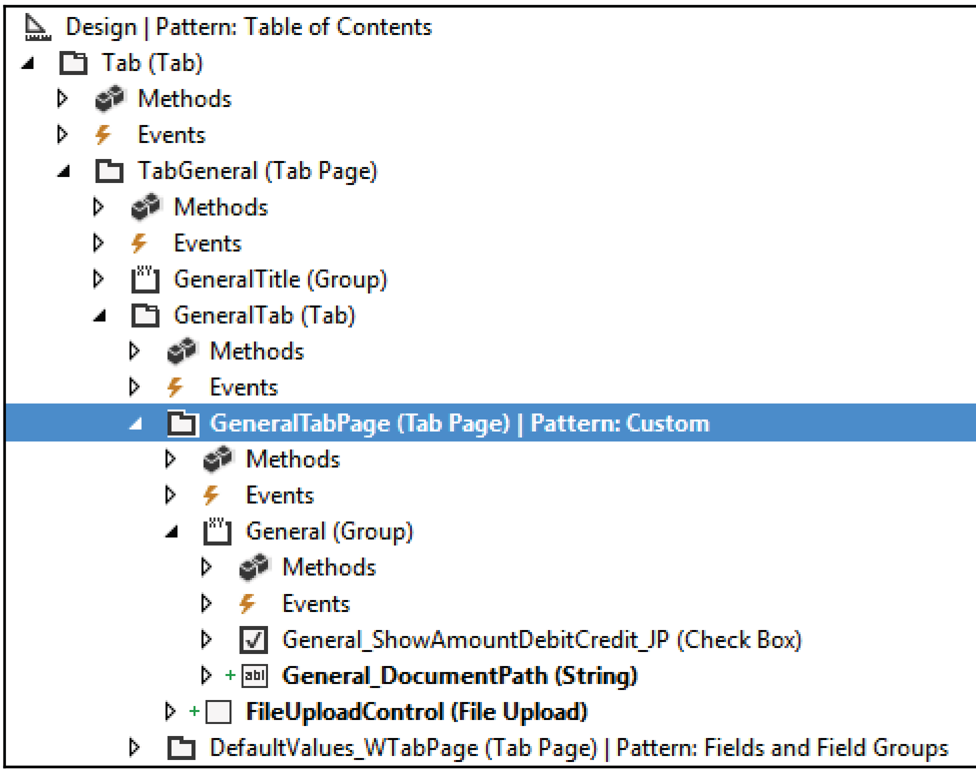
1. Create a **new project**. Open AOT, add a `VendParameters` table for customization, and create a new field with the following properties:

Property	Value
Type	String
Name	DocumentPath
Label	Documents
ExtendedDataType	FilePath

2. Add the newly created field to the bottom of the table's `General` field group.
3. In AOT, find form `VendParameters` and add it to the project using the customize option.
4. Select form and datasource `VendParameters` and select the `Restore` option.
5. Add a new `FileUploadControl` next to the Document path with the following properties:



6. After restore, the form design should look as follows:



7. Next, open the `VendParameters` form and change the following methods:
    - `init()`
    - `closeOk()`

8. Declare a variable in the `init` method as follows:

```
FileUpload  uploadControl;
```

9. Add the following lines in the `init()` method:

```
uploadControl = fileuploadControl;
uploadControl.notifyUploadCompleted +=
eventhandler(FileUploadControl.uploadCompleted);
```

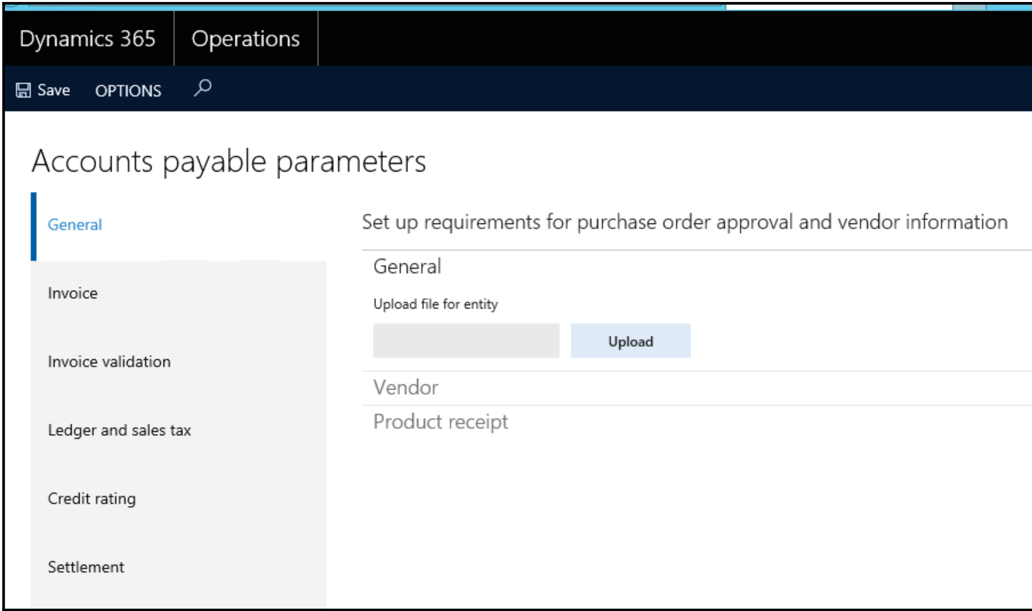
10. Add new method `closeOk()` on the `VendParameters` form as follows:

```
 public void closeOk()
{
   FileUpload  uploadControl;
   uploadControl = FileUploadControl;
   uploadControl.notifyUploadCompleted -=
   eventhandler(FileUploadControl.uploadCompleted);
   super();
}
```

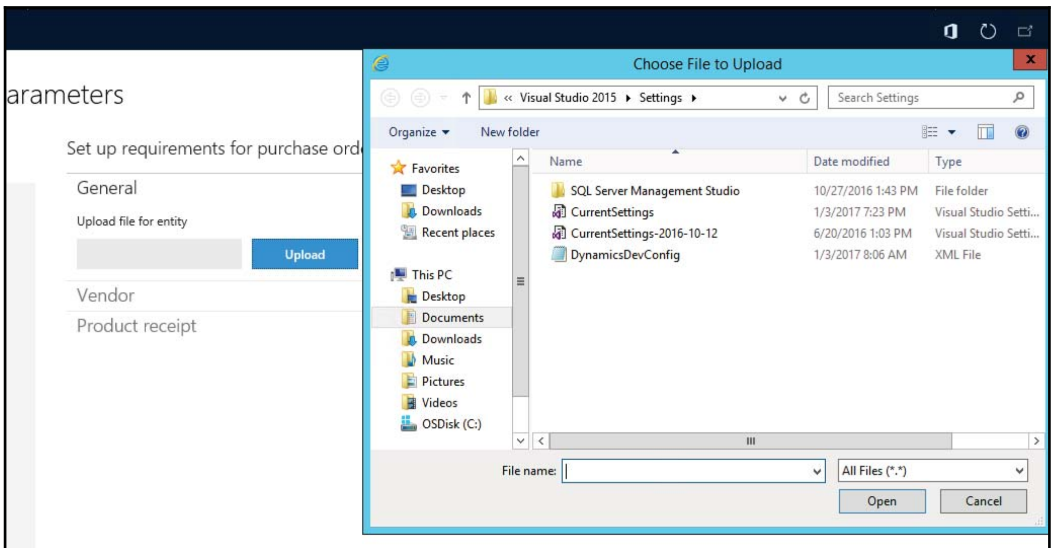
11. Add a new method on `FileUploadControl` and add `new code`:

```
 public void uploadCompleted()
{
  FileUploadTemporaryStorageResult    fileUploadResult =
  FileUploadControl.getFileUploadResult();
  if (fileUploadResult != null &&
  fileUploadResult.getUploadStatus())
 {
    VendParameters.DocumentPath =
fileUploadResult.getFileName();
 }
}
```

12. Build and synchronize the project.
13. As a result, we will be able to select and store a text file in the **Accounts receivables** | **Setup** | **Accounts receivables parameters** form in the **Upload file** field under the **General** tab page:



14. In the preceding screen, when you click on the **Upload** button under the **General** group, a dialog opens up where we need to choose the file to upload, as shown in the following screenshot:



How it works...

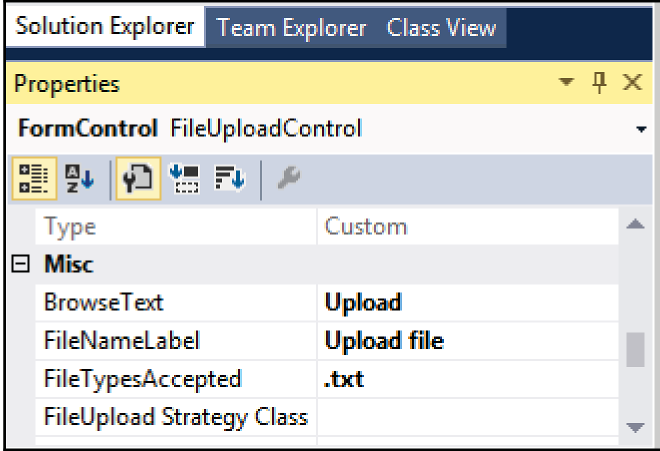
In this recipe, we first create a new field to store the file location. We use the `Filepath` extended data type. We also add this field to the field group in the table to ensure that it is displayed on the form automatically. A `File upload` control is added to handle file upload events on the form.

The following form methods are called by the `file upload` control and must be present on the caller form:

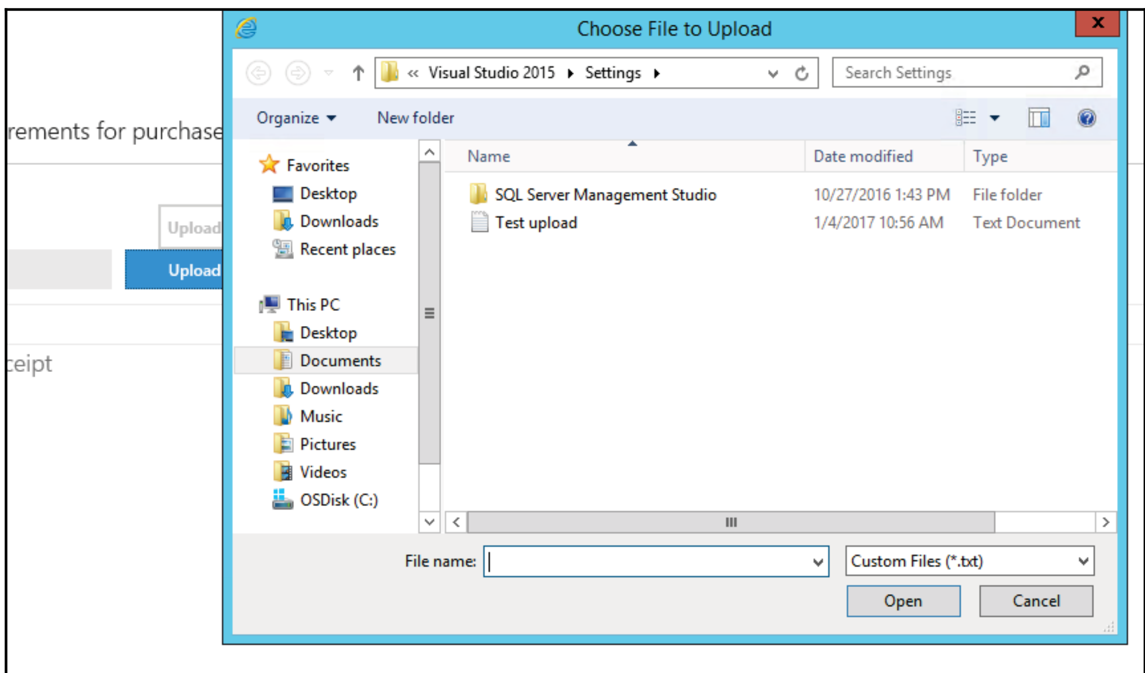
- The `uploadCompleted()` method contains code to get the file path and place it in the `DocumentPath` field
- The `int()` and `closeOK()` method delegates the `uploadCompleted()` method

There's more...

Additionally, if we want to select a file of a certain type, then we can easily go on `FileUploadControl`, look at its properties, and find `FileTypesAccepted`. We could select `.txt` as shown in the following screenshot:



This would make our **browse folder** lookup as follows, and would by default allow us to select a `*.txt` file:



Creating a color picker lookup

In Dynamics 365 for Finance and Operations, the **color selection dialog boxes** are used in various places, allowing the user to select and store a color code in a **table** field. Then the stored color code can be used in various places to color data records, change form backgrounds, set colors for various controls, and so on.

In this recipe, we will create a **color** lookup. For demonstration purposes, we will add an option to set a color for each legal entity in the system.

How to do it...

1. In the AOT, open the `CompanyInfo` table and create a new field with the following properties:

Property	Value
Type	Integer
Name	CompanyColor
ExtendedDataType	CCColor

2. Open the `OMLegalEntity` form, locate the `TopPanel` group in **Design** | **Tab** | **General**, and add a new `IntEdit` control with the following properties to the bottom of the group:

Property	Value
Name	CompanyColor
AutoDeclaration	Yes
LookupButton	Always
ShowZero	No
Label	Company color

3. In the same form, create a new method with the following code snippet in the `CompanyInfo` data source:

```
public edit CCColor editCompanyColor(boolean     _set,
CompanyInfo _companyInfo,
CCColor     _color)
{
    if (_companyInfo.CompanyColor)
    {
        CompanyColor.backgroundColor(_companyInfo.CompanyColor);
    }
    else
    {
CompanyColor.backgroundColor(WinAPI::RGB2int(255,255,255));
    }
CompanyColor.foregroundColor(CompanyColor.backgroundColor());
    return _companyInfo.CompanyColor;
```

```
        }
```

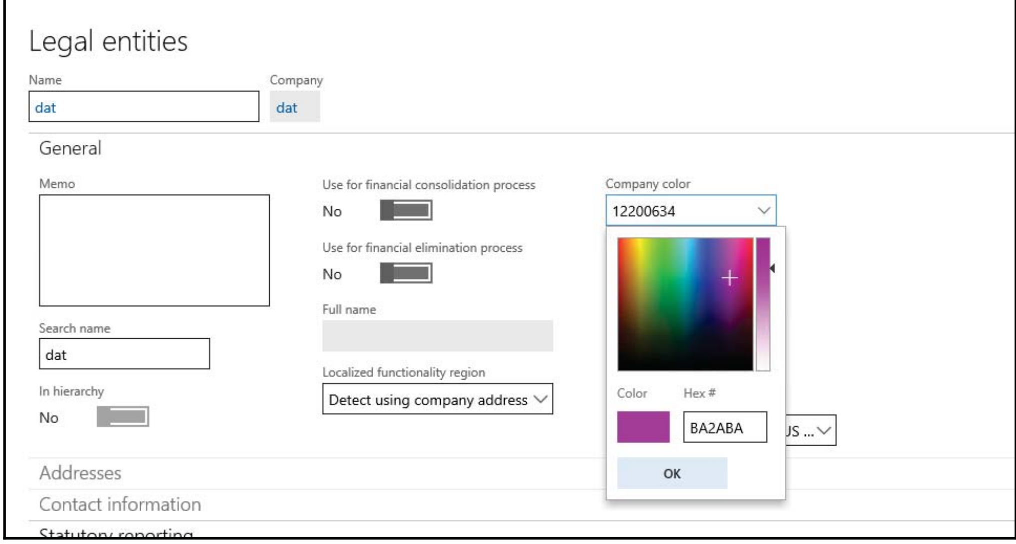
4. Update the properties of the newly created `CompanyColor` control as follows:

Property	Value
DataSource	CompanyInfo
DataMethod	editCompanyColor

5. On the same control, override its `lookup()` method with the following code snippet:

```
public void lookup()
{
    int         red;
    int         green;
    int         blue;
    int color = this.value();
    color = ColorSelection::selectColor(this, color);
    CompanyInfo.CompanyColor = color;
    this.value(color);
    this.backgroundColor(color);
}
```

6. To test the results, navigate to **Organization administration** | **Organization** | **Legal entities** and note the newly created **Company color** lookup:



How it works...

Dynamics 365 for Finance and Operations does not have a special control to select colors. Therefore, we have to create a fake control, which is presented to the user as a color selection.

Colors in Dynamics 365 for Finance and Operations are stored as integers, so we first create a new `Integer` field on the `CompanyInfo` table. On the form, we create a new control, which will display the color. The created control does not have any automatic lookup and therefore it does not have the lookup button next to it. We have to force the button to appear by setting the control's `LookupButton` property to `Always`.

Next, we create a `new edit` method, which is then set on the created control as a `data` method. This method is responsible for changing the control's background to match the stored color. This gives an impression to the user that the chosen color was saved. The background is set to `white` if no value is present. The method always returns the value `0` because we do not want to show the actual color code in it. The control's `ShowZero` property is set to `No` to ensure that even the returned `0` is not displayed. In this way, we create a control that looks like a real `color selection` control.

The last thing to do is to override the control's `lookup()` method with the code that invokes the **color selection dialog box**. Here, we use the `selectColor` method of the `ColorSelection` class to convert the current control's background color into a **red-green-blue component set**. This set is then passed to the `value()` method to make sure that the currently set color is selected on the lookup initially. The `selectColor()` method is the `main` method, which invokes the lookup. It accepts the following arguments:

- The current window handle
- A `binary object` representing up to 16 custom colors

This method returns an integer code of the color components, which has to be converted back to a numeric value in order to store it in the table field.

5

Processing Business Tasks

In this chapter, we will cover the following recipes:

- Using a segmented entry control
- Creating a general journal
- Posting a general journal
- Processing a project journal
- Creating and posting a ledger voucher
- Changing an automatic transaction text
- Creating a purchase order
- Posting a purchase order
- Creating a sales order
- Posting a sales order
- Creating an electronic payment format

Introduction

In Dynamics 365 for Finance and Operations, various business operations, such as creating financial journals, posting sales orders, and generating vendor payments are performed from the user interface by users on a periodic basis. For developers, it is very important to understand how it works internally in new Dynamics 365 for Finance and Operations so that the logic can be used to design and implement new customized business logic.

This chapter will explain how various Dynamics 365 business operations can be performed through the code. We will discuss how to perform different operations on various journals, sales order, purchase orders, and so on. This chapter also explains how to work with the ledger voucher object and how to enhance the setup of the automatically-generated transaction texts. Posting purchase and sales orders and changing business document layout per company are also discussed here. This chapter includes other features, such as creating a new electronic payment format and controlling the display of inventory dimensions.

Using a segmented entry control

In Dynamics 365 for Finance and Operations, segmented entry control can simplify the task of entering complex account and dimension combinations. The control consists of a dynamic number of elements, named segments. The number of segments may vary depending on the setup, and their lookup values may depend on the values specified in other segments in the same control. The segmented entry control always uses the controller class, which handles the entry and display in the control.

In this recipe, we will show you how a segmented entry control can be added to a form. In this demonstration, we will add a new `Ledger account` control to the **general ledger parameters** form, assuming that the control can be used as a default ledger account for various functions. The example does not make much sense in practice, but it is perfectly suitable to demonstrate the usage of the segmented entry control.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Create a new extension of the `LedgerParameters` table in your project and create a new `Int64` type field with the following properties (click on **Yes** to automatically add a foreign key relationship once you are asked):

Property	**Value**
Name	LedgerDimension
ExtendedDataType	LedgerDimensionAccount

2. Add the newly created field to the `General` group in the table.

3. Find the table's relation, named `DimensionAttributeValueCombination`, and change its property, as follows:

Property	Value
UseDefaultRoleNames	No

4. In the project, add the `LedgerParameters` form and declare the following variables in class declaration:

```
MainAccountRecId         currentMainAccountId;
MainAccountRecId         previousMainAccountId;
MainAccountRecId         currentOffsetMainAccountId;
DimensionAttributeRecId  mainAccountDimAttr;
LedgerJournalEngine      ledgerJournalEngine;
```

5. In the same form, find the `General_LedgerDimension` segmented entry control by going to **Tab** | **LedgerTab** | **LedgerTabFastTab** | **GeneralTabPage** | **General**, and then change the field properties:

Property	Value
Auto Declaration	Yes
Controller class	DimensionDynamicAccountController
Include Financial accounts	Yes
Is default account	False

6. Now override three of its methods with the following code snippet:

```
public void onSegmentChanged(DimensionControlSegment _segment)
{
    if (_segment.parmDimensionAttribute().RecId ==
    mainAccountDimAttr)
     {
        previousMainAccountId = currentMainAccountId;
     }
    super(_segment);
    ledgerJournalEngine =
    LedgerJournalEngine::construct(LedgerJournalType::Daily,
    element);
    ledgerJournalEngine.ledgerJournalTable
     (element.args().record());
}
```

7. Add the following lines of code at the bottom of the form's `init()` method before `super()`:

```
mainAccountDimAttr =
DimensionAttribute::getWellKnownDimensionAttribute
(DimensionAttributeType::MainAccount);
```

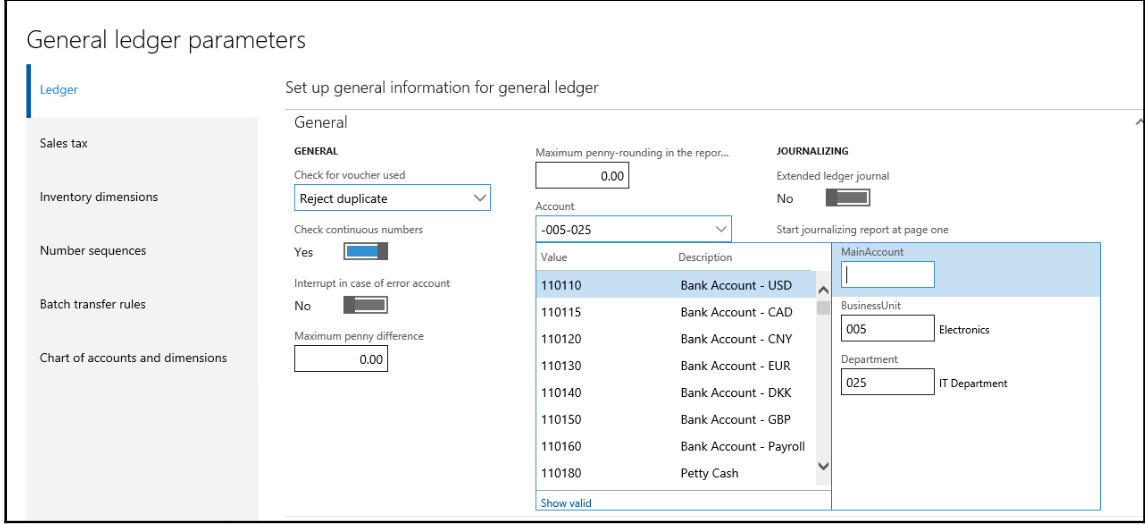
8. In the `active()` method of `datasource LedgerParameters`, add the following line of code:

```
currentMainAccountId =
MainAccount::getMainAccountRecIdFromLedgerDimension
(LedgerParameters.LedgerDimension);
previousMainAccountId = currentMainAccountId;
```

9. Add the `DimensionHierarchyHelper` class to the project and add a few lines of code in the `getHierarchyTypeByAccountType()` method at line number 442 under `case enumNum(LedgerJournalACType)`:

```
default :
return  DimensionHierarchyType::AccountStructure;
```

10. To test the results, navigate to **General ledger** | **Setup** | **General ledger parameters** and notice the newly created `Ledger account` control, which allows you to select and save the main account and a number of financial dimensions, as shown in the following screenshot:



How it works...

We start the recipe by creating a new field in the `LedgerParameters` table. The field extends the `LedgerDimensionAccount` extended data type in order to ensure that the segmented entry control appears automatically, once this field is added to the user interface. We also add the newly created field to one of the table's groups in order to make sure that it appears on the form automatically.

Next, we have to modify the `LedgerParameters` form. In its class declaration and the `init()` method, we define and instantiate the `LedgerDimensionAccountController` class, which handles the events raised by the segmented entry control. The combination of the class and the control allows the user to see a dynamic number of segments, based on the system configuration.

Then, we override the following methods in the control:

- `loadAutoCompleteData()`: This retrieves the autocompleted data
- `loadSegments()`: This loads the value stored in the table field into the control
- `segmentedValueChanged()`: This updates the controller class when the value of the control is changed by the user

Lastly, we override the following methods in the data source field:

- `resolveReference()`: This finds the ledger account record specified by the user
- `jumpRef()`: This enables the **View details** link in the control's right-click context menu
- `validate()`: This performs user input validation

There's more...

In this section, we will discuss how the input of the segmented entry control can be simulated from the code. It is very useful when migrating or importing data into the system. In the Dynamics Project, add the `DimensionAttributeValueCombination` table and create a new method with the following code snippet:

```
    public static LedgerDimensionAccount getLedgerDimension(
    MainAccountNum _mainAccountId,
    container      _dimensions,
    container      _values)
  {
     MainAccount                       mainAccount;
```

```
 DimensionHierarchy              dimHier;
 LedgerStructure                 ledgerStruct;
 Map                             dimSpec;
 Name                            dimName;
 Name                            dimValue;
 DimensionAttribute              dimAttr;
 DimensionAttributeValue         dimAttrValue;
 List                            dimSources;
 DimensionDefaultingEngine       dimEng;
 int                             i;
 mainAccount = MainAccount::findByMainAccountId(
  _mainAccountId);
if (!mainAccount.RecId)
{
   return 0;
}
select firstOnly RecId from dimHier
where dimHier.StructureType ==
DimensionHierarchyType::AccountStructure
&& dimHier.IsDraft == NoYes::No
exists join ledgerStruct
where ledgerStruct.Ledger == Ledger::current()
&& ledgerStruct.DimensionHierarchy == dimHier.RecId;
if (!dimHier.RecId)
{
   return 0;
}
 dimSpec =
 DimensionDefaultingEngine::createEmptyDimensionSpecifiers();
 for (i = 1; i <= conLen(_dimensions); i++)
{
   dimName = conPeek(_dimensions, i);
   dimValue = conPeek(_values, i);
   dimAttr = DimensionAttribute::findByName(dimName);
  if (!dimAttr.RecId)
  {
      continue;
  }
    dimAttrValue =
    DimensionAttributeValue::findByDimensionAttributeAndValue(
    dimAttr, dimValue, false, true);
    if (dimAttrValue.IsDeleted)
    {
       continue;
    }
    DimensionDefaultingEngine::insertDimensionSpecifer(
    dimSpec,
    dimAttr.RecId,
```

```
            dimValue,
            dimAttrValue.RecId,
            dimAttrValue.HashKey);
        }
        dimSources = new List(Types::Class);
        dimSources.addEnd(dimSpec);
        dimEng = DimensionDefaultingEngine::constructForMainAccountId(
        mainAccount.RecId,
        dimHier.RecId);
        dimEng.applyDimensionSources(dimSources);
        return dimEng.getLedgerDimension();
    }
```

This method can be used to convert a combination of main accounts and a number of financial dimension values into a ledger account. The method accepts the following three arguments:

- The main account number
- A container of dimension names
- A container of dimension values

We start this method by searching for the main account record. We also locate the record of the hierarchy of the current chart of accounts.

Next, we fill an empty map with the dimension values. Before inserting each value, we check whether the dimension and its value are present in the system. To do this, we use the methods in the `DimensionAttribute` and `DimensionAttributeValue` tables to do.

We end the method by creating a new `DimensionDefaultingEngine` object and passing the list of dimensions and their values to it. Now, when everything is ready, the `getLedgerDimension()` method of `DimensionDefaultingEngine` returns the ledger account number.

See also

- The *Creating a general journal* recipe
- The *Creating and posting a ledger voucher* recipe

Creating a general journal

Journals in Dynamics 365 for Finance and Operations are manual worksheets that can be posted into the system. One of the frequently used journals for financial operations is the general journal. It allows the virtual processing of any type of posting: ledger account transfers, fixed asset operations, customer/vendor payments, bank operations, project expenses, and so on. Journals, such as the f**ixed assets journal**, p**ayment journal** in **Accounts receivable** or **Accounts payable**, and many others, are optimized for specific business tasks, but they basically do the same job.

In this recipe, we will demonstrate how to create a new general journal record from the code. The journal will hold a single line for debiting one ledger account and crediting another one. For demonstration purposes, we will specify all the input values in the code.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Create a new Dynamics 365 solution named `CreateGeneralJournal`. Change the model name in properties with the one created earlier.
2. In the project, create a new class named `LedgerJournalTransData` with the following code snippet:

```
public class LedgerJournalTransData extends JournalTransData
{
}
public void create(
boolean _doInsert        = false,
boolean _initVoucherList = true)
{
  lastLineNum++;
  journalTrans.LineNum = lastLineNum;
 if (journalTableData.journalVoucherNum())
  {
     this.initVoucher(
      lastVoucher,
       false,
       _initVoucherList);
  }
    this.addTotal(false, false);
   if (_doInsert)
   {
     journalTrans.doInsert();
```

```
        }
        else
       {
          journalTrans.insert();
       }
       if (journalTableData.journalVoucherNum())
       {
         lastVoucher = journalTrans.Voucher;
       }
     }
```

3. Add the `LedgerJournalStatic` class in your project and replace its `newJournalTransData()` method with the following code snippet:

```
   JournalTransData newJournalTransData(
    JournalTransMap  _journalTrans,
     JournalTableData _journalTableData)
 {
    return new LedgerJournalTransData(
     _journalTrans,
      _journalTableData);
 }
```

4. Create a new class named `GetLedgerDimension` with the following code snippet:

```
   class GetLedgerDimension
  {
     public static LedgerDimensionAccount  getLedgerDimension(
      MainAccountNum _mainAccountId,
       container      _dimensions,
        container      _values)
    {
       MainAccount                    mainAccount;
       DimensionHierarchy             dimHier;
       LedgerStructure                ledgerStruct;
       Map                            dimSpec;
       Name                           dimName;
       Name                           dimValue;
       DimensionAttribute             dimAttr;
       DimensionAttributeValue        dimAttrValue;
       List                           dimSources;
       LedgerDimensionDefaultingEngine      dimEng;
       int                            i;
       mainAccount = MainAccount::findByMainAccountId(
       _mainAccountId);
      if (!mainAccount.RecId)
```

```
 {
    return 0;
 }
 select firstOnly RecId from dimHier
 where dimHier.StructureType ==
 DimensionHierarchyType::AccountStructure
 && dimHier.IsDraft == NoYes::No
 exists join ledgerStruct
 where ledgerStruct.Ledger == Ledger::current()
 && ledgerStruct.DimensionHierarchy ==  dimHier.RecId;
if (!dimHier.RecId)
{
   return 0;
}
 dimSpec = LedgerDimensionDefaultingEngine::
  createEmptyDimensionSpecifiers();
 for (i = 1; i <= conLen(_dimensions); i++)
{
   dimName = conPeek(_dimensions, i);
   dimValue = conPeek(_values, i);
   dimAttr = DimensionAttribute::findByName(dimName);
  if (!dimAttr.RecId)
  {
     continue;
  }
   dimAttrValue =
   DimensionAttributeValue::findByDimensionAttributeAndValue(
   dimAttr, dimValue, false, true);
  if (dimAttrValue.IsDeleted)
  {
     continue;
  }
   LedgerDimensionDefaultingEngine::insertDimensionSpecifer(
   dimSpec,
   dimAttr.RecId,
   dimValue,
   dimAttrValue.RecId,
   dimAttrValue.HashKey);
}
 dimSources = new List(Types::Class);
 dimSources.addEnd(dimSpec);
 dimEng = LedgerDimensionDefaultingEngine::
  constructForMainAccountId(
   mainAccount.RecId,
    dimHier.RecId);
 dimEng.applyDimensionSources(dimSources);
 return dimEng.getLedgerDimension();
}
```

```
}
```

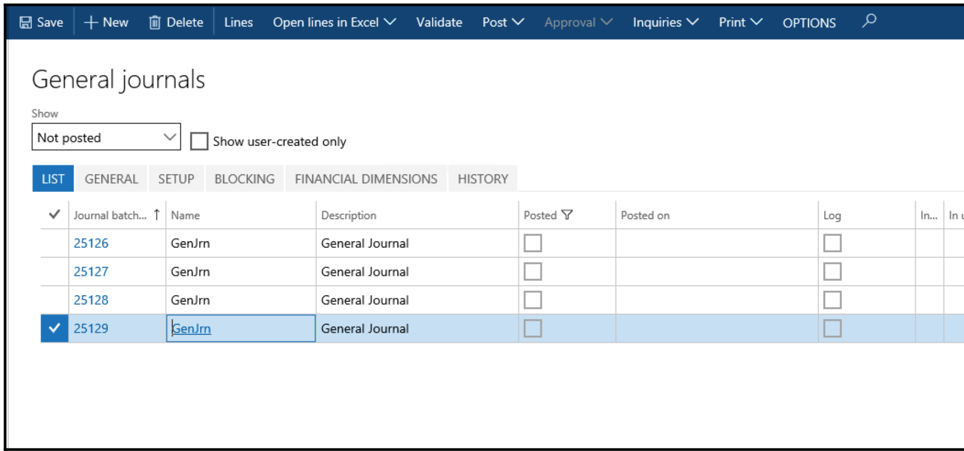
5. Create another class named `LedgerJournalCreate` with the following code snippet:

```
class LedgerJournalCreate
{
  public static void Main(Args _args)
 {
    LedgerJournalTable      jourTable;
    LedgerJournalTrans      jourTrans;
    LedgerJournalTableData  jourTableData;
    LedgerJournalTransData  jourTransData;
    LedgerJournalStatic     jourStatic;
    DimensionDynamicAccount ledgerDim;
    DimensionDynamicAccount offsetLedgerDim;
    ttsBegin;
    ledgerDim =
     GetLedgerDimension::getLedgerDimension(
      '110180',
       ['BusinessUnit', 'Department'],
        ['005', '024']);
    offsetLedgerDim =
    GetLedgerDimension::getLedgerDimension(
     '170150',
      [' BusinessUnit', 'Department'],
       ['005', '024']);
    jourTableData = JournalTableData::newTable(jourTable);
    jourTable.JournalNum = jourTableData.nextJournalId();
    jourTable.JournalType = LedgerJournalType::Daily;
    jourTable.JournalName = 'GenJrn';
    jourTableData.initFromJournalName(
     LedgerJournalName::find(jourTable.JournalName));
    jourStatic    = jourTableData.journalStatic();
    jourTransData = jourStatic.newJournalTransData(
     jourTrans,
      jourTableData);
    jourTransData.initFromJournalTable();
    jourTrans.CurrencyCode          = 'USD';
    jourTrans.initValue();
    jourTrans.TransDate             = systemDateGet();
    jourTrans.LedgerDimension       = ledgerDim;
    jourTrans.Txt                   = 'General journal demo';
    jourTrans.OffsetLedgerDimension = offsetLedgerDim;
    jourTrans.AmountCurDebit        = 1000;
    jourTransData.create();
    jourTable.insert();
    ttsCommit;
```

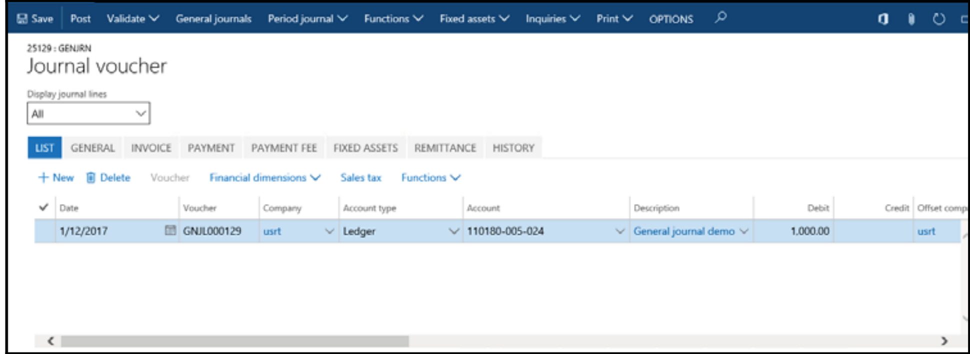
```
        info(strFmt(
        "Journal '%1' has been created", jourTable.JournalNum));
    }
  }
```

6. Save all your code and set this class as `set as start up object`. Now, run the project and you will get the following message:

7. Now check the results by navigating to **General ledger** | **Journal entries** | **General journals**, as shown in the following screenshot:



8. Click on the **Lines** button to open journal lines and notice the created line, as shown in the following screenshot:



How it works...

We start the recipe by creating the `LedgerJournalTransData` class, which will handle the creation of journal lines. It inherits everything from the `JournalTransData` class, apart from its `create()` method. Actually, this method is a copy of the same method from the `JournalTransData` class, with the exception that it does not contain the code that is not relevant to the ledger journal creation. We also modify the `newJournalTransData()` constructor of the `LedgerJournalStatic` class to use our newly created class.

The journal creation code is placed in a new job. We start the code by initializing ledger dimensions. Here, we use the `getLedgerDimension()` method from the previous recipe to get ledger dimensions. This method accepts three parameters: the main account number, a container of dimension names, and a container of dimension values. In this example, the ledger dimensions consist of the main account, business unit, and department, and its value is `110180-005-024`. Use your own values depending on the data you have.

We also create a new `jourTableData` object that is used for journal record handling. Then, we set the journal number, type, and name and call the `initFromJournalName()` method to initialize some additional values from the journal name settings. At this stage, the journal header record is ready.

Next, we create a journal line. We create a new `jourTransData` object to handle the journal line, and we call its `initFromJournalTable()` method to initialize additional values from the journal header. Then, we set some of the journal line values, such as the currency and transaction date.

Finally, we call the `create()` method on the `jourTransData` object and the `insert()` method on the `jourTable` object to create the journal line and header records, respectively. The journal is now ready to be reviewed.

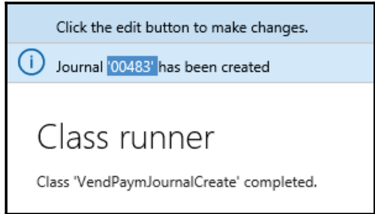
There's more

The preceding example can be easily modified to create different journals, not just the general journal. For instance, the **payment journal** in the **Accounts payable** module is based on the same data sources as the general journal and some of its code is the same. So, let's create a new, similar job named `VendPaymJournalCreate` with the following code snippet:

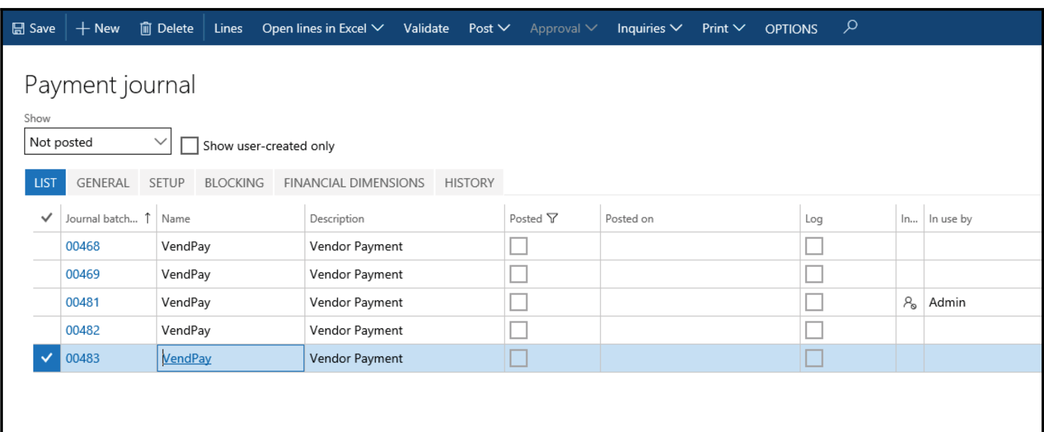
```
 class VendPaymJournalCreate
{
   public static void Main(Args _args)
```

```
    {
        LedgerJournalTable      jourTable;
        LedgerJournalTrans      jourTrans;
        LedgerJournalTableData  jourTableData;
        LedgerJournalTransData  jourTransData;
        LedgerJournalStatic     jourStatic;
        DimensionDynamicAccount ledgerDim;
        DimensionDynamicAccount offsetLedgerDim;
        ttsBegin;
        ledgerDim = LedgerDynamicAccountHelper:
         :getDynamicAccountFromAccountNumber('1001',
          LedgerJournalACType::Vend);
        LedgerJournalACType::Vend);
        offsetLedgerDim = LedgerDynamicAccountHelper:
         :getDynamicAccountFromAccountNumber(
          'USMF OPER',
           LedgerJournalACType::Bank);
        //Journal header data
        jourTableData = JournalTableData::newTable(jourTable);
        jourTable.JournalNum  = jourTableData.nextJournalId();
        jourTable.JournalType = LedgerJournalType::Payment;
        jourTable.JournalName = 'VendPay';
        jourTableData.initFromJournalName(
         LedgerJournalName::find(jourTable.JournalName));
        jourStatic    = jourTableData.journalStatic();
        //Journal line data
        jourTransData = jourStatic.newJournalTransData(
         jourTrans,
          jourTableData);
        jourTransData.initFromJournalTable();
        jourTrans.CurrencyCode    = 'USD';
        jourTrans.initValue();
        jourTrans.TransDate       = systemDateGet();
        jourTrans.AccountType     = LedgerJournalACType::Vend;
        jourTrans.LedgerDimension = ledgerDim;
        jourTrans.Txt             = 'Vendor payment journal   demo';
        jourTrans.OffsetAccountType     = LedgerJournalACType::Bank;
        jourTrans.OffsetLedgerDimension = offsetLedgerDim;
        jourTrans.AmountCurDebit        = 1000;
        jourTransData.create();
        jourTable.insert();
        ttsCommit;
        info(strFmt(
         "Journal '%1' has been created", jourTable.JournalNum));
    }
  }
```

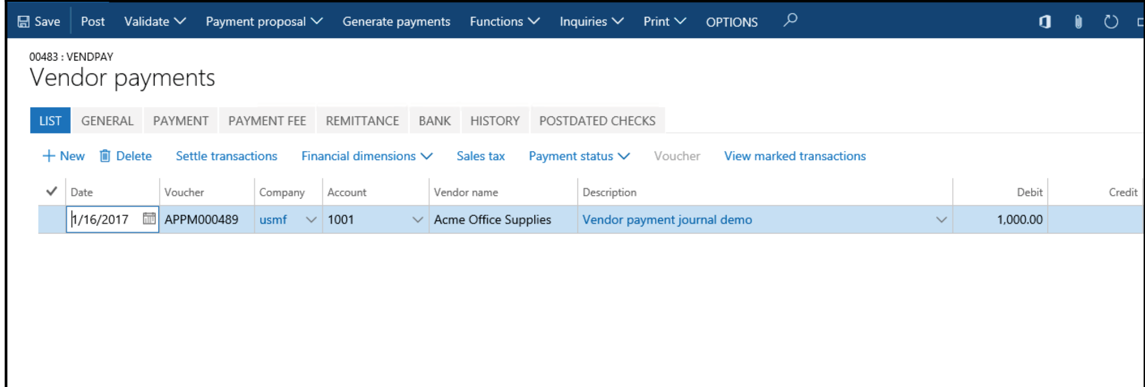
When you run your code, your output will look as follows:



Now, the newly created journal can be found by navigating to **Accounts payable** | **Journals** | **Payments** | **Payment journal**, as shown here:



The journal's lines should reflect what we've specified in the code, as shown in the following screenshot:



The code in this section has only slight differences compared to the previous example, as follows:

- The ledger dimension contains a reference to a vendor account, and the offset ledger dimension refers to a bank account record
- The journal type is changed to a vendor disbursement, that is, `LedgerJournalType::Payment`
- The journal name to be matched with the payment journal configuration is different
- The journal line account type is set to `vendor`, and the offset account type is set to `bank`

See also

- The *Using a segmented entry control* recipe
- The *Posting a general journal* recipe

Posting a general journal

Journal posting is the next step once the journal has been created. Although most of the time journals are posted from the user interface, it is also possible to perform the same operation from the code.

In this recipe, we will explore how a general journal can be posted from the code. We are going to process the journal created in the previous recipe.

How to do it...

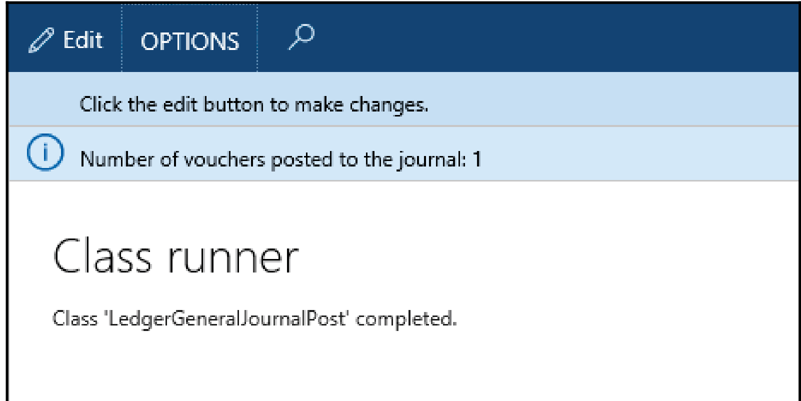
Carry out the following steps in order to complete this recipe:

1. Navigate to **General ledger** | **Journals** | **General journal** and find an open journal. Create a new journal if none exists. Note the journal's number.

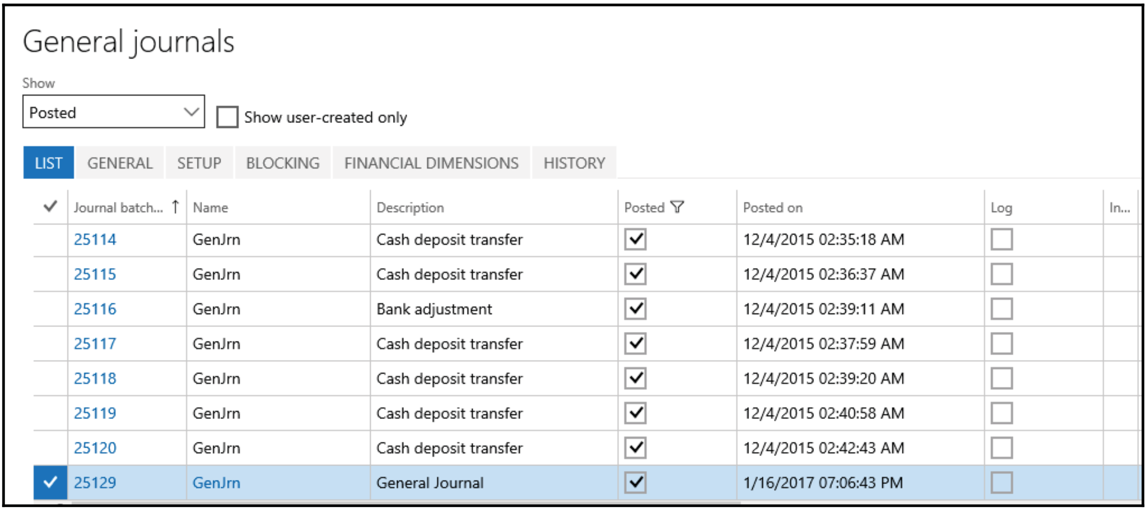
2. In your solution, add a new runnable class named `LedgerJournalPost` with the following code snippet (replace the `00472` text with the journal's number from the previous step):

```
 static void LedgerJournalPost(Args _args)
{
   LedgerJournalCheckPost LedgerJournalCheckPost;
   LedgerJournalTable     LedgerJournalTable;

   LedgerJournalTable = LedgerJournalTable::find('00472');
   LedgerJournalCheckPost=
    LedgerJournalCheckPost::newLedgerJournalTable(
     jourTable,
      NoYes::Yes);

   LedgerJournalCheckPost.run();
}
```

3. Save all your code and build your solution.
4. Now, test set this class as a startup object, run the solution, and notice the **Infolog** window, confirming that the journal was successfully posted, as shown here:



5. Navigate to **General ledger** | **Journals** | **General journal** and locate the journal in order to make sure that it was posted, as shown in the following screenshot:



How it works...

In this recipe, we created a new job named `LedgerGeneralJournalPost`, which holds all the code. Here, we use the `LedgerJournalCheckPost` class, which does all the work. This class ensures that all the necessary validations are performed. It also locks the journal so that no user can access it from the user interface.

In the job, we create the `jourPost` object by calling the `newLedgerJournalTable()` constructor on the `LedgerJournalCheckPost` class. This method accepts a journal header record to be processed and a second argument, defining whether the journal should be validated and posted or only validated. In this recipe, we find the previously created journal record and pass it to the `LedgerJournalCheckPost` class along with the second argument, instructing it to perform both validation and posting.

See also

- The *Creating a general journal* recipe

Processing a project journal

As with most of the modules in Dynamics 365 for Finance and Operations, the **Project management and accounting** module contain several journals, such as **hour**, **expense**, **fee**, and **item**. Although they are similar to the **general journal**, they provide a more convenient user interface to work with projects and contain some module-specific features.

In this recipe, we will create and post a project journal from the code. We will process an **hour** journal, holding a registered employee's time.

How to do it...

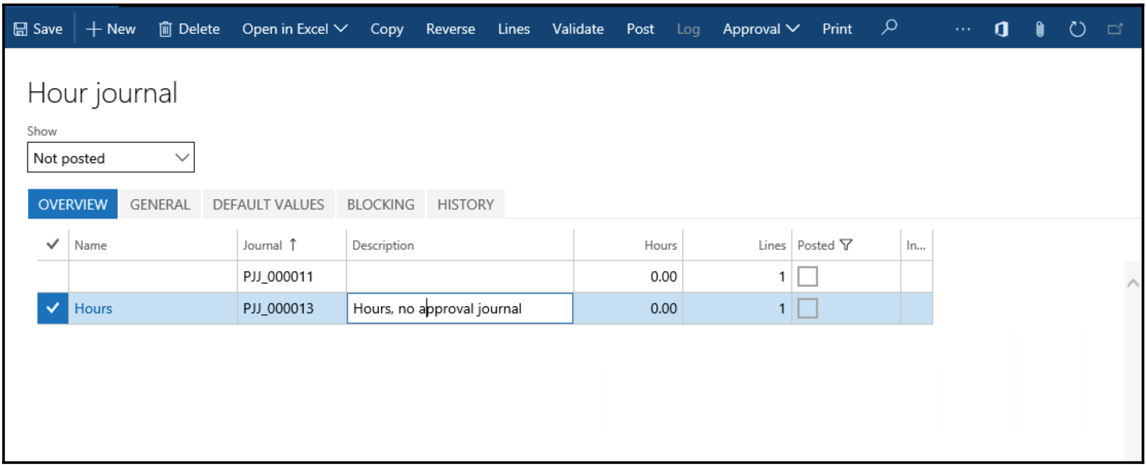
Carry out the following steps in order to complete this recipe:

1. Create a new project `ProcessProjectJournal`, and assign our custom model to it.
2. Create a new class named `ProjJournalCreate` with the following code snippet (replace the input values in the code to match your data):

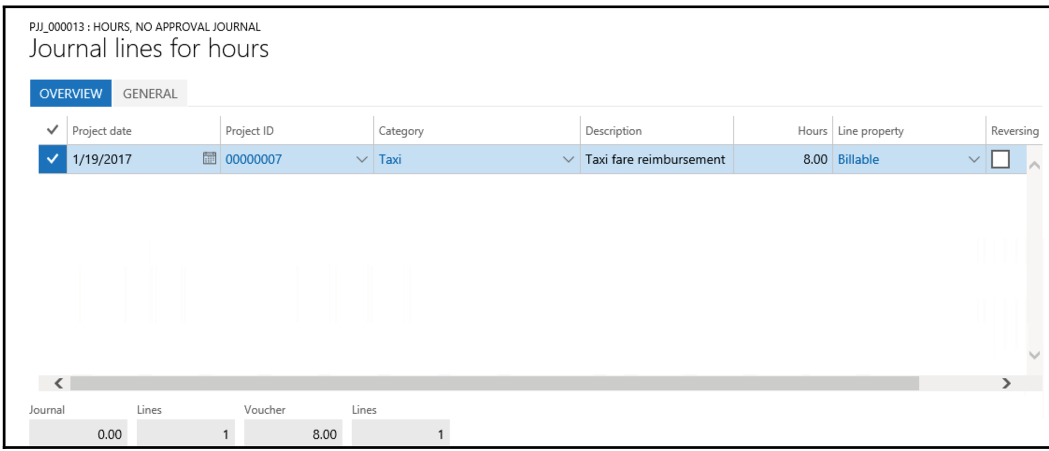
```
 class ProjJournalCreate
{
  public static void Main(Args _args)
 {
    ProjJournalTable     jourTable;
    ProjJournalTrans     jourTrans;
    ProjJournalTableData jourTableData;
    ProjJournalTransData jourTransData;
    ProjJournalStatic    jourStatic;
    ttsBegin;
    jourTableData = JournalTableData::newTable(jourTable);
    jourTable.JournalId     = jourTableData.nextJournalId();
    jourTable.JournalType   = ProjJournalType::Hour;
    jourTable.JournalNameId = 'Hours';
    jourTableData.initFromJournalName(
     ProjJournalName::find(jourTable.JournalNameId));
    jourStatic = jourTableData.journalStatic();
    jourTransData = jourStatic.newJournalTransData(
     jourTrans,
      jourTableData);
    jourTransData.initFromJournalTable();
    jourTrans.initValue();
    jourTrans.ProjId = '00000007';
    jourTrans.initFromProjTable(
     ProjTable::find(jourTrans.ProjId));
```

```
            jourTrans.TransDate     = systemDateGet();
            jourTrans.ProjTransDate = jourTrans.TransDate;
            jourTrans.CategoryId = 'Taxi';
            jourTrans.setHourCostPrice();
            jourTrans.setHourSalesPrice();
            jourTrans.TaxItemGroupId =
            ProjCategory::find(jourTrans.CategoryId).TaxItemGroupId;
            jourTrans.DEL_Worker =
             HcmWorker::findByPersonnelNumber('000062').RecId;
            jourTrans.Txt = 'Taxi fare reimbursement';
            jourTrans.Qty = 8;
            jourTransData.create();
            jourTable.insert();
            ttsCommit;
            info(strFmt(
            "Journal '%1' has been created", jourTable.JournalId));
        }
    }
```

3. Execute the class and check the results by navigating to **Project management and accounting** | **Journals** | **Hour**, as shown in the following screenshot:



4. Click on the **Lines** button to open journal lines and notice the newly created record, as shown in the following screenshot:



How it works...

In this recipe, we create a new job where we store all the code. In the job, we use the `ProjJournalTableData` and `ProjJournalTransData` classes in a way similar to how we used the `LedgerJournalTableData` and `LedgerJournalTransData` classes in the *Creating a general journal* recipe. Here, we create a new `jourTableData` object used for journal record handling. Then, we initialize the journal number, type, and name of the actual journal record. For demonstration purposes, we set the journal name in the code, but it can be easily replaced with a value from some parameter. Next, we call `initFromJournalName()` on the `jourTableData` object in order to initialize some additional values from the journal name settings. At this stage, the journal header record is ready.

Next, we create a journal line. Here, we first create a new `jourTransData` object to handle the journal line. Then, we call its `initFromJournalTable()` method in order to initialize the additional values from the journal header. Finally, we set some of the journal line values, such as **transaction** and **project date, category**, and **worker number**. Normally, these values have to be taken from the user input, external data, or any other source, depending on the functionality being built. In this example, we simply specify the values in the code.

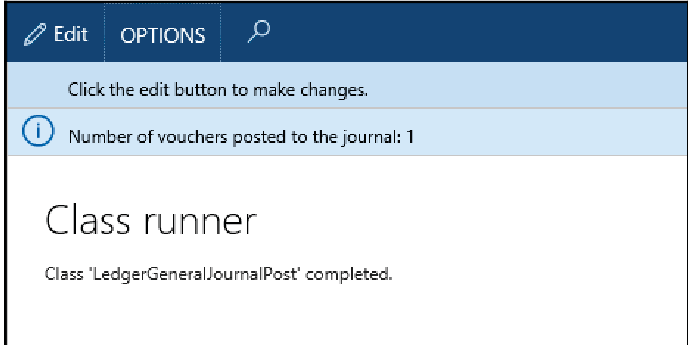
Lastly, we call the `create()` method on `jourTransData` and the `insert()` method on `jourTable` to create the journal line and the header records, respectively. The journal is now ready to be reviewed.

There's more...

For further journal processing, we can use the class named `ProjJournalCheckPost` to post project journals from the code. In the Dynamics project, let's create another class named `ProjJournalPost` with the following code snippet (replace `PJJ_000013` with your journal number):

```
class ProjJournalPost
{
  public static void Main(Args _args)
  {
    ProjJournalCheckPost jourPost;
    jourPost =   ProjJournalCheckPost::newJournalCheckPost(
    true,
    true,
    JournalCheckPostType::Post,
    tableNum(ProjJournalTable),
    'PJJ_000013');
    jourPost.run();
  }
}
```

Run the job to post the journal. The **Infolog** window should display the confirmation, as shown here:



In the newly created job, we use the `newJournalCheckPost()` constructor of the `ProjJournalCheckPost` class. The constructor accepts the following arguments:

- A **Boolean** value that specifies whether to block the journal while it is being posted or not. It is a good practice to set the value to `true`, as this ensures that no one modifies this journal while it is being posted.
- A Boolean value that specifies whether to display results in the **Infolog** window.

- The type of action being performed. The possible values for this class are either `Post` or `Check`. The latter one only validates the journal, and the first one validates and posts the journal at once.
- The table ID of the journal being posted.
- The journal number to be posted.
- Finally, we call the `run()` method, which posts the journal.

Creating and posting a ledger voucher

In Dynamics 365 for Finance and Operations, all the financial transactions, regardless of where they are originated, end up in the **General ledger** module. When it comes to customized functionality, developers should use the **Dynamics 365 APIs** to create the required system entries. No transactions can be created directly in the tables, as this may affect the accuracy of financial data.

In order to ensure data consistency, the system provides numerous APIs for developers to use. One of them is **ledger voucher processing**. This allows you to post a financial voucher in the **General ledger** module. Vouchers in Dynamics 365 for Finance and Operations are balanced financial entries that represent a single operation. They include two or more ledger transactions. The ledger voucher API ensures that all the mandatory fields, such as voucher numbers, ledger accounts, offset account, financial dimensions, balances, and others, are filled and valid.

In this recipe, we will demonstrate how a ledger voucher can be created and posted from the code. We will create a single voucher with two balancing transactions.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Double-check whether the `getLedgerDimension()` method exists in the `DimensionAttributeValueCombination` table. If not, create it as described in the first recipe of this chapter.

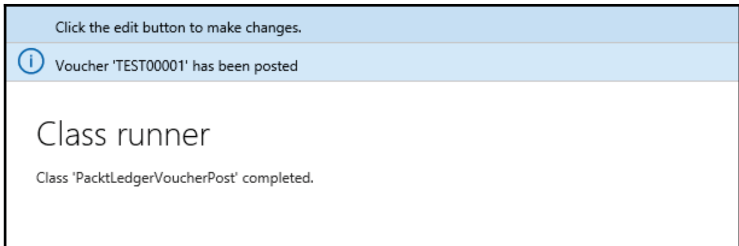
2. In the solution, create a new job named `LedgerVoucherPost` with the following code snippet:

```
 class PacktLedgerVoucherPost
{
```

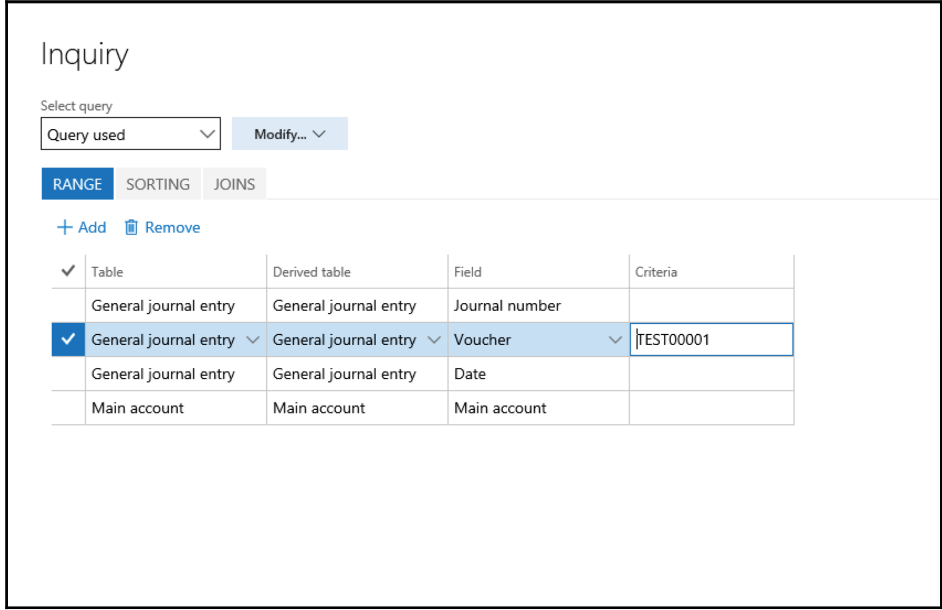
```
 public static void Main(Args _args)
{
   LedgerVoucher            LedgerVoucher;
   LedgerVoucherObject      voucherObj;
   LedgerVoucherTransObject voucherTrObj1;
   LedgerVoucherTransObject voucherTrObj2;
   DimensionDynamicAccount  ledgerDim;
   DimensionDynamicAccount  offsetLedgerDim;
   CurrencyExchangeHelper   currencyExchHelper;
   CompanyInfo              companyInfo;
   ledgerDim =
    GetLedgerDimension::getLedgerDimension(
    '110180',
     ['BusinessUnit', 'Department'],
      ['005', '024']);
   offsetLedgerDim =
    GetLedgerDimension::getLedgerDimension(
    '170150',
     ['BusinessUnit', 'Department'],
      ['005', '024']);
   LedgerVoucher  = LedgerVoucher::newLedgerPost(
    DetailSummary::Detail,
     SysModule::Ledger,
      '');
   voucherObj = LedgerVoucherObject:
    :newVoucher('TEST00001');
   companyInfo = CompanyInfo::findDataArea(curext());
   currencyExchHelper =
    CurrencyExchangeHelper::newExchangeDate(
   Ledger::primaryLedger(companyInfo.RecId),
   voucherObj.parmAccountingDate());
   LedgerVoucher.addVoucher(voucherObj);
   voucherTrObj1 =
   LedgerVoucherTransObject::newTransactionAmountDefault(
    voucherObj,
     LedgerPostingType::LedgerJournal,
      ledgerDim,
      'USD',
       1000,
        currencyExchHelper);
   voucherTrObj2 =
   LedgerVoucherTransObject::newTransactionAmountDefault(
    voucherObj,
     LedgerPostingType::LedgerJournal,
      offsetLedgerDim,
       'USD',
        -1000,
         currencyExchHelper);
```

```
            LedgerVoucher.addTrans(voucherTrObj1);
            LedgerVoucher.addTrans(voucherTrObj2);
            LedgerVoucher.end();
            info(strFmt(
             "Voucher '%1' has been posted", voucher.lastVoucher()));
        }
    }
```

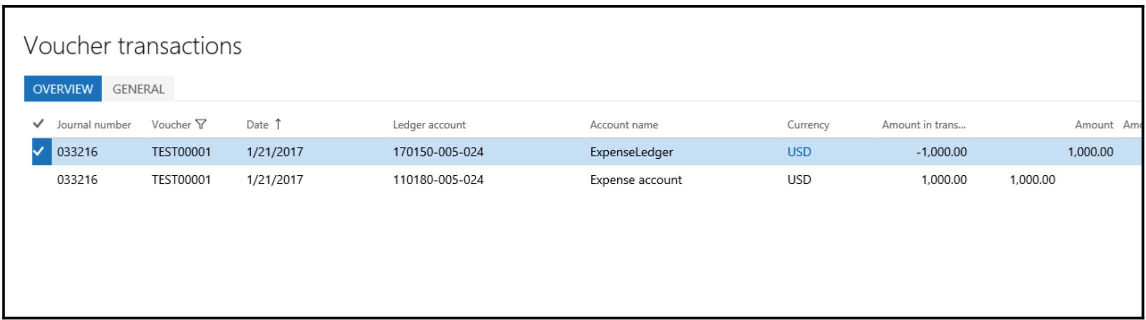
3. Run the class to create a new ledger voucher, as shown in the following screenshot:



4. To check what has been posted, navigate to **General Ledger** | **Inquiries** | **Voucher transactions** and type in the voucher number used in the code, as shown in the following screenshot:



5. Click on **OK** to display the posted voucher:



How it works...

In the newly created job, we first define the ledger accounts where the posting will be done. Normally, this comes from the user input, but for demonstration purposes, here we have specified it in the code. We use the previously created `getLedgerDimension()` method to simulate the ledger account entry.

Next, we create a new `LedgerVoucher` object, which represents a collection of vouchers. Here, we call the `newLedgerPost()` constructor of the `LedgerVoucher` class. The `newLedgerPost()` constructor accepts three mandatory and four optional arguments, which are listed as follows:

- Post detailed or summarized ledger transactions.
- The system module from which the transactions originate.
- A number sequence code, which is used to generate the voucher number. In this example, we will set the voucher number manually. So, this argument can be left empty.
- The transaction type that will appear in the transaction log.
- The transaction text.

- A Boolean value, which specifies whether this voucher should meet the approval requirements.
- A Boolean value, defining whether the voucher can be posted without a posting type when posting inventory transactions.

Then, we create a new `LedgerVoucherObject` object, which represents a single voucher. We call the `newVoucher()` constructor of the `LedgerVoucherObject` class. It accepts only one mandatory parameter and a number of optional parameters, which are listed as follows:

- The voucher number; normally, this should be generated using a number sequence, but in this example, we set it manually
- The transaction date; the default is the `session` date
- The system module from which the transactions originate
- The ledger transaction type
- A flag defining whether this is a correcting voucher; the default is `No`
- The posting layer; the default is `Current`
- The document number
- The document date
- The acknowledgement date
- The `addVoucher()` method of the `LedgerVoucher` class adds the created voucher object to the voucher

Once the voucher is ready, we create two voucher transactions. The transactions are handled by the `LedgerVoucherTransObject` class. They are created by calling its `newTransactionAmountDefault()` constructor with the following mandatory arguments:

- The ledger voucher object
- The ledger posting type
- The ledger account number
- The currency code
- The amount in the currency
- The currency exchange rate helper

Notice the last argument, which is a currency exchange rate helper, used when operating in currencies other than the main company currency.

We add the created transaction objects to the voucher by calling its `addTrans()` method. At this stage, everything is ready for posting.

Finally, we call the `end()` method on the `LedgerVoucher` object, which posts the transactions to the ledger.

See also

- The *Using a segmented entry control* recipe

Changing an automatic transaction text

Every financial transaction in Dynamics 365 for Finance and Operations must have a descriptive text. Some texts are entered by users and some can be generated by the system. The latter option holds true for automatically generated transactions, where the user cannot interact with the process.

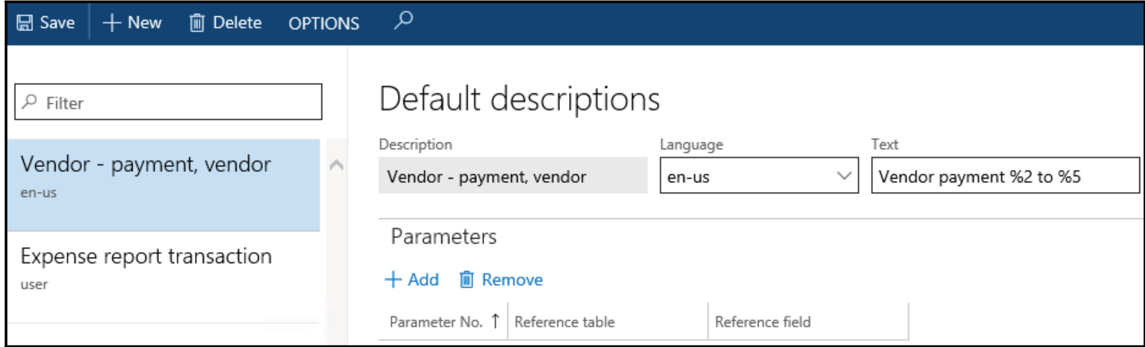
Dynamics 365 for Finance and Operations provides a way to define texts for automatically generated transactions. The setup can be found by navigating to **Organizations administration** | **Setup** | **Default descriptions**. Here, the user can create custom transaction texts for various automatic transaction types and languages. The text itself can have a number of placeholders--digits with a percent sign in front of them, which are replaced with actual values during the process. Placeholders can be from `%1` to `%6`, and they can be substituted with the following values:

- `%1`: This is the transaction date
- `%2`: This is a relevant number, such as the invoice and delivery note
- `%3`: This is the voucher number
- `%4` to `%6`: This is custom and depends on the module

In this recipe, we will demonstrate how the existing automatic transaction text functionality can be modified and extended. One of the places where it is used is the automatic creation of vendor payment journal lines, during the vendor payment proposal process. We will modify the system so that the texts of the automatically-generated vendor payment lines include the vendor names.

Getting ready

First, we need to make sure that the vendor payment transaction text is set up properly. Navigate to **Organization administration** | **Setup** | **Default descriptions**, find a line with **Vendor - payment, vendor**, (if this record is not there, you can create a new one), and change the text to `Vendor payment %2 to %5`, as shown in the following screenshot:



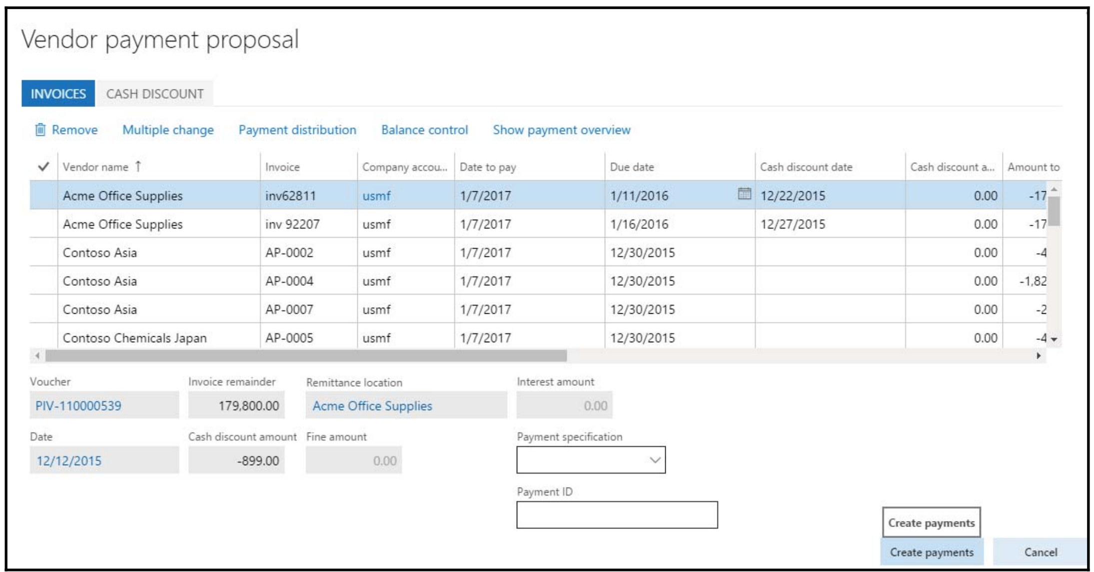
How to do it...

Carry out the following steps in order to complete this recipe:

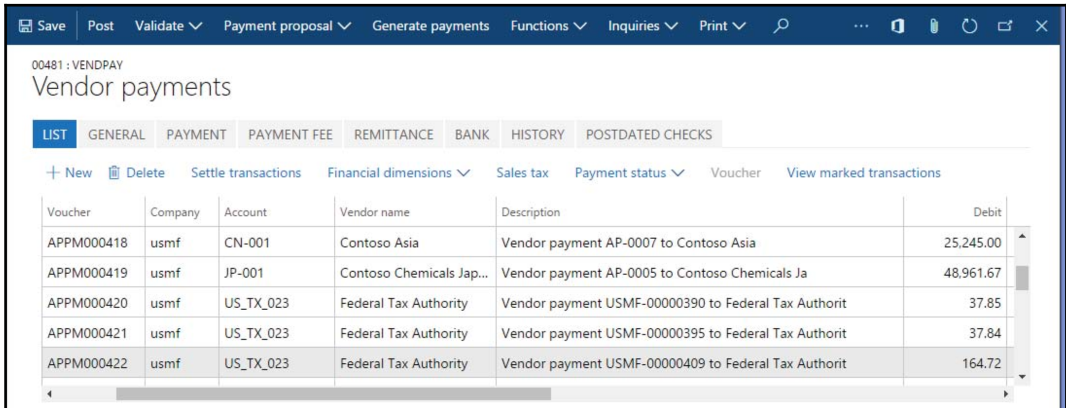
1. Add the `CustVendPaymProposalTransferToJournal` class to your project and add the following lines of code at the bottom of the `getTransactionText()` method, right before its return:

```
transactionTxt.setKey2(
 _custVendPaymProposalLine.custVendTable().name());
```

2. Navigate to **Accounts payable** | **Payments** | **Payment journal** and create a new journal. Open **journal lines,** run **Create payment proposal**, which is under **Payment proposal**, from the action pane. Define the desired criteria or leave the field blank and click on **OK**. In the newly opened **Vendor payment proposal** form, click on the **Create Payment** button to transfer all the proposed lines to the journal. See the following screenshot:



3. Notice that the transaction text in each journal line includes the vendor name, as shown in the following screenshot:



How it works...

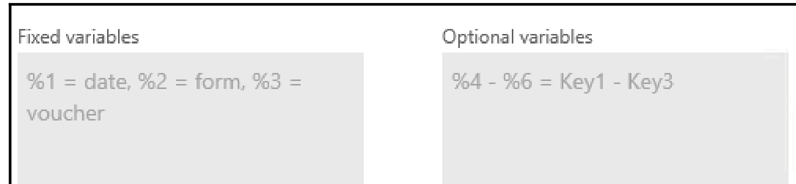
The vendor payment proposal uses the `CustVendPaymProposalTransferToJournal` class to create the lines. The same class contains a method named `getTransactionText()`, which is responsible for formatting the text in each line. If we look inside it, we can see that the `TransactionTxt` class is used for this purpose. This class contains the following methods, which are used to substitute the placeholders from `%1` to `%6` in the defined text:

- `%1`: `setDate()`
- `%2`: `setFormLetter()`
- `%3`: `setVoucher()`
- `%4`: `setKey1()`
- `%5`: `setKey2()`
- `%6`: `setKey3()`

By taking a look at the code, you can see that only the `%4` placeholder is used. So, you can fill the `%5` placeholder with the vendor name. To achieve this, you need to call the `setKey2()` method with the vendor name as an argument. In this way, every journal line created by the automatic vendor payment proposal will contain a vendor name in its description.

There's more...

In standard application, we have limited placeholders, as shown in the following screenshot:



If more than three custom placeholders are required, it is always possible to add an additional placeholder, by creating a new `setKey()` method in the `TransactionTxt` class. For example, if we want to add a `%7` placeholder, we have to do the following:

1. Add the following line of code to the class declaration of the `TransactionTxt` class:

    ```
    str 20 key4;
    ```

2. Create a new method with the following code snippet:

    ```
    void setKey4(str 20 _key4)
    {
       key4 = _key4;
    }
    ```

3. Change the last line of the `txt()` method to the following:

    ```
    return strFmt(
      txt,
      date2StrUsr(transDate, DateFlags::FormatAll),
      formLetterNum,
      voucherNum,
      key1,
      key2,
      key3,
      key4);
    ```

4. Now, we can use the `setKey4()` method to substitute the `%7` placeholder.

Note that, although more placeholders can be added, you should take into consideration the fact that the transaction text field has a finite number of characters and excessive text will simply be truncated.

Creating a purchase order

Purchase orders are used throughout the purchasing process to hold information about the goods or services that a company buys from its suppliers. Normally, purchase orders are created from the user interface, but in automated processes, purchase orders can be also created from the code.

In this recipe, you will learn how to create a purchase order from the code. We will use a standard method provided by the application.

How to do it...

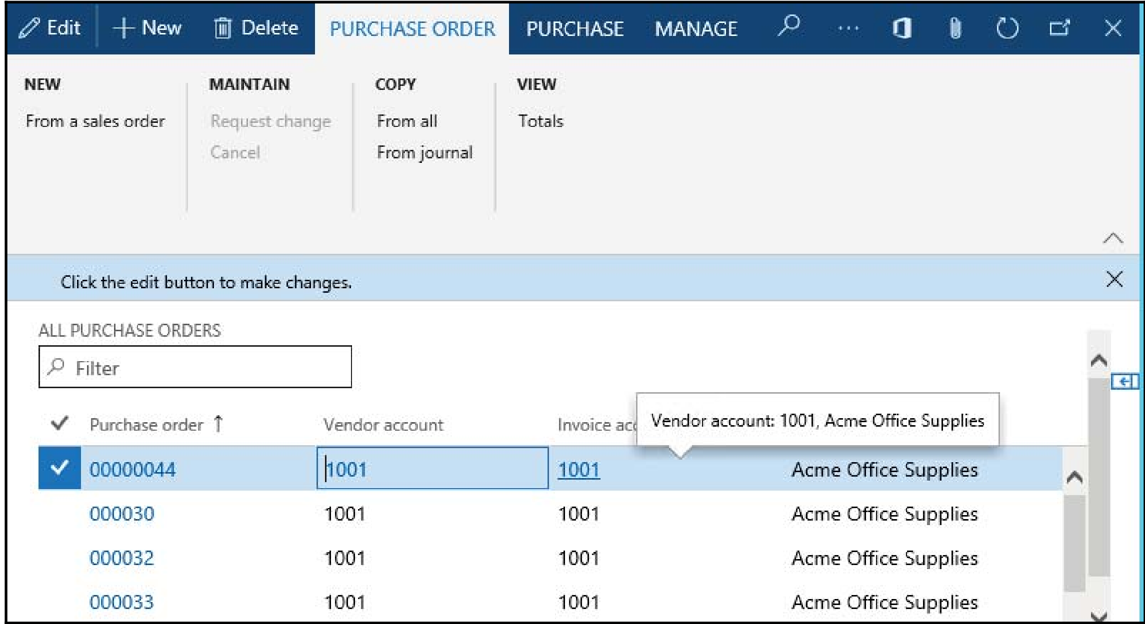
Carry out the following steps in order to complete this recipe:

1. Add a new runnable class named `CreatePurchOrder` with the following code snippet:

```
 static void PktCreatePurchOrder(Args _args)
{
   NumberSeq  numberSeq;
   PurchTable purchTable;
   PurchLine  purchLine;

   ttsBegin;
   //initialize number sequence objects
   numberSeq = NumberSeq::newGetNum(
    PurchParameters::numRefPurchId());
     numberSeq.used();
   purchTable.PurchId = numberSeq.num();
   purchTable.initValue();
   //Initialize new record in PurchTable using vendor account
   purchTable.initFromVendTable(VendTable::find('vend001'));
   if (!purchTable.validateWrite())
  {
     throw Exception::Error;
  }
   purchTable.insert();
   //insert purchase line
   purchLine.PurchId = purchTable.PurchId;
   purchLine.ItemId  = 'item001';
   purchLine.createLine(true, true, true, true, true, true);
   ttsCommit;
   info(strFmt("New Purchase order '%1' has been created",
   purchTable.PurchId));
}
```

2. Save and build your code and select this class as **set as startup object**. Now run the project to create a new purchase order.

3. Navigate to **Procurement and sourcing** | **Common** | **Purchase orders** | **All purchase orders** in order to view the purchase order created, as shown in the following screenshot:



How it works...

In this recipe, we created a new job named `CreatePurchOrder`, which holds all the code. Here, we start by getting the next purchase order number with the help of the `NumberSeq` class. We also call the `initValue()` and `initFromVendTable()` methods to initialize various `purchTable` buffer fields. Normally, the argument of the `initFromVendTable()` method should come from a user selection screen or some other source, but for demonstration purposes, we specify the value in the code. We insert the purchase order record into the table only if the validation in the `validateWrite()` method is successful.

Next, we create purchase order lines. Here, we assign the previously used purchase order number and then set the item number. As previously mentioned, such values should come from a user input or some other source, but for demonstration purposes, we specify it in the code.

Finally, we call the `createLine()` method of the `PurchLine` table to create a new line. This is a very useful method, allowing you to quickly create purchase order lines. This method accepts a number of optional Boolean arguments, which are listed as follows:

- Perform data validations before saving; the default is `false`
- Initialize the line record from the `PurchTable` table; the default is `false`
- Initialize the line record from the `InventTable` table; the default is `false`
- Calculate inventory quantity; the default is `false`
- Add miscellaneous charges; the default is `true`
- Use trade agreements to calculate the item price; the default is `false`
- Do not copy the inventory site and warehouse from the purchase order header; the default is `false`
- Use purchase agreements to get the item price; the default is `false`

There's more...

You can also use the data entities, to insert **Purchase Order Header** and **Line** records. To insert purchase order header, use the `PurchPurchaseOrderHeaderEntity` entity and for Purchase Order Line data use the `PurchPurchaseOrderLineEntity` data entity.

In the preceding code sample, we used a few methods to set some mandatory values in `PurchTable` and `PurchLine`. Until you find similar methods in these entities, you may have to assign all mandatory values manually.

Posting a purchase order

In Dynamics 365 for Finance and Operations, the purchase order goes through a number of statuses in order to reflect its current position within the purchasing process. The status can be updated either manually by using the user interface or programmatically from the code as well.

In this recipe, we will demonstrate how a purchase order status can be updated from the code. We will confirm the purchase order created in the previous recipe and print the relevant document on the screen.

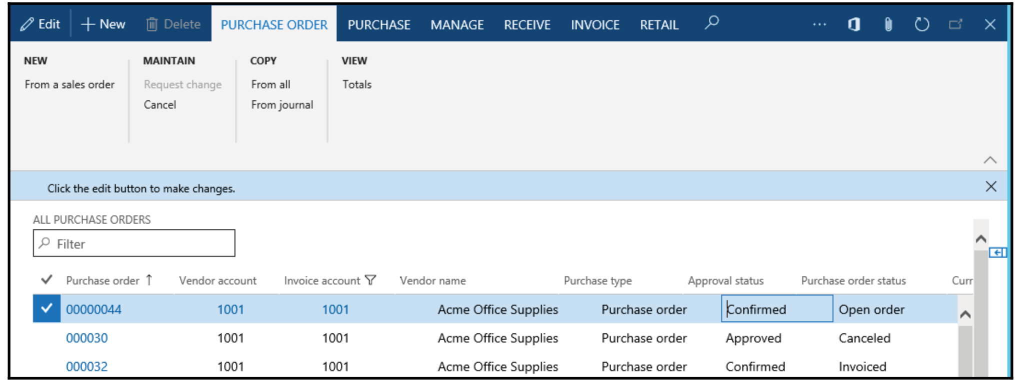
How to do it...

Carry out the following steps in order to complete this recipe:

1. Add a new runnable class, named `ConfirmPurchOrder` with the following code snippet. Replace `00000044` with your number, that is created after previous code `CreatePurchOrder` (your PO number could be different from mine so double-check):

```
 static void ConfirmPurchOrder (Args _args)
{
   PurchFormLetter purchFormLetter;
   PurchTable      purchTable;
   purchTable = PurchTable::find('00000044');
   purchFormLetter = PurchFormLetter::construct(
    DocumentStatus::PurchaseOrder);
   purchFormLetter.update(
    purchTable,
   '',
   DateTimeUtil::date(DateTimeUtil::utcNow()),
   PurchUpdate::All,
   AccountOrder::None,
   NoYes::No,
   NoYes::Yes);
}
```

2. Save and build your code and select this class as **set as startup object**. Now, run the project to post the specified purchase order.
3. Navigate to **Procurement and sourcing** | **Common** | **Purchase orders** | **All purchase orders** and note that the **Approval status** column of the posted order is now different, as shown here:



How it works...

In this recipe, we create a new job named `ConfirmPurchOrder`, which holds all the code.

First, we find a purchase order, which we are going to update. In this recipe, we use the purchase order created in the previous recipe. Here, we will normally replace the code with a user input or an output from some other function.

Next, we create a new `PurchFormLetter` object using its `construct()` constructor. The constructor accepts an argument of the `DocumentStatus` type, which defines the type of posting to be done. Here, we use `DocumentStatus::PurchaseOrder` as a value, as we want to confirm the purchase order.

The last thing to do is to call the `update()` method of the `PurchFormLetter` object, which does the actual posting. It accepts a number of arguments, which are listed as follows:

- The purchase order header record; in this case, it is the `PurchTable` table.
- An external document number; it's not used in this demonstration, as it is not required when posting a purchase order confirmation.
- The transaction date; the default date is the system's date.
- The quantity to be posted; the default is `PurchUpdate::All`. Other options, such as `PurchUpdate::PackingSlip` or `PurchUpdate::ReceiveNow`, are not relevant when confirming a purchase order.
- The order summary update; this argument is not used at all. The default is `AccountOrder::None`.
- A Boolean value defining whether a preview or the actual posting should be done.
- A Boolean value defining whether the document should be printed.
- A Boolean value specifying whether printing management should be used. The default value is `false`.
- A Boolean value defining whether to keep the remaining quantity on order; otherwise, it is set to zero. This argument is used when posting credit notes.
- A container of a number of `TmpFrmVirtual` records. This argument is optional and is used only when posting purchase invoices.

There's more...

The same technique can be used to post a purchase packing slip, invoice, or update to any other status, which is available in a given context. Let's take a look at the following example:

```
purchFormLetter = PurchFormLetter::construct(
DocumentStatus::PurchaseOrder);
```

Replace the preceding code snippet with the following:

```
purchFormLetter = PurchFormLetter::construct(
DocumentStatus::Invoice);
```

Now, let's take another code snippet:

```
purchFormLetter.update(
purchTable,
'',
DateTimeUtil::date(DateTimeUtil::utcNow()),
PurchUpdate::All,
AccountOrder::None,
NoYes::No,
NoYes::Yes);
```

Replace the preceding code snippet with the following:

```
purchFormLetter.update(
purchTable,
'8001',
DateTimeUtil::date(DateTimeUtil::utcNow()),
PurchUpdate::All,
AccountOrder::None,
NoYes::No,
NoYes::Yes);
```

Now, when you run the job, the purchase order will be updated to an invoice. To check the updated purchase order, navigate to **Procurement and sourcing** | **Common** | **Purchase orders** | **All purchase orders**; notice that its **Status** field is different now.

If you are adding your objects in new projects, then you may need to set this project property **set as startup project** as well, to run your preceding code while you run the whole solution/project.

Creating a sales order

Sales orders are used throughout the sales process to hold information about the goods or services that a company sells to its customers. Normally, sales orders are created from the user interface, but for the automated processes, sales orders can also be created from the code.

In this recipe, you will learn how to create a sales order from the code. We will use a standard method provided by the application.

How to do it...

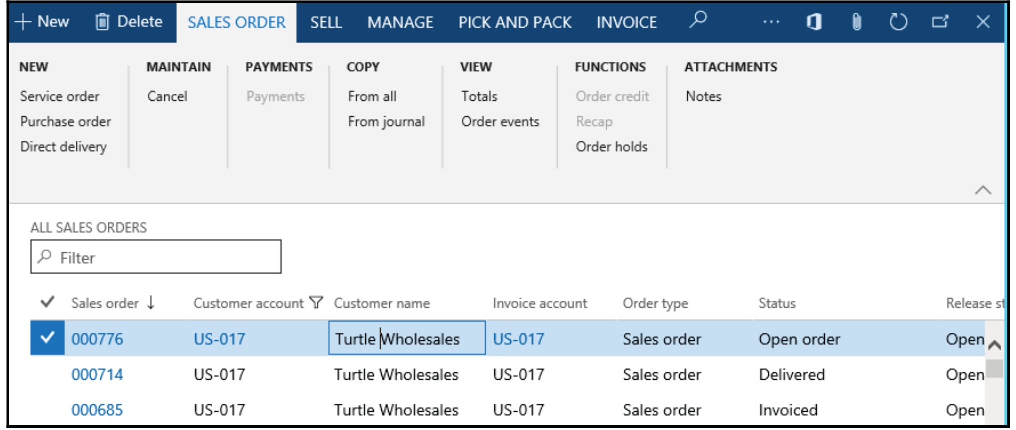
Carry out the following steps in order to complete this recipe:

1. Add a new `Runnable` class in your project and name it `SalesOrderCreate`. Copy and paste the following code in the main method of this class:

```
static void SalesOrderCreate(Args _args)
{
    NumberSeq  numberSeq;
    SalesTable salesTable;
    SalesLine  salesLine;
    ttsBegin;
    numberSeq = NumberSeq::newGetNum(
    SalesParameters::numRefSalesId());
    numberSeq.used();
    salesTable.SalesId = numberSeq.num();
    salesTable.initValue();
    salesTable.CustAccount = 'US-017';
    salesTable.initFromCustTable();
    if (!salesTable.validateWrite())
   {
      throw Exception::Error;
   }
    salesTable.insert();
    salesLine.SalesId = salesTable.SalesId;
    salesLine.ItemId  = 'D0001';
    salesLine.createLine(true, true, true, true, true, true);
    ttsCommit;
    info(strFmt(
     "Sales order '%1' has been created", salesTable.SalesId));
}
```

2. Save and build your code and select this class as **set as startup object**. Now, run the project to create a new sales order.

3. Navigate to **Sales and marketing** | **Common** | **Sales orders** | **All sales orders** in order to view the newly created sales order, as shown in the following screenshot:



How it works...

In this recipe, we create a new job named `SalesOrderCreate`, which holds all the code. The job starts by generating the next sales order number with the help of the `NumberSeq` class. We also call the `initValue()` and `initFromCustTable()` methods to initialize various `salesTable` buffer fields. Notice that, for `initFromCustTable()`, we first set the customer account and call the method afterwards, instead of passing the customer record as an argument. We insert the sales order record into the table only if the validation in the `validateWrite()` method is successful.

Next, we create sales order lines. Here, we assign the previously used sales order number and set the item number.

Finally, we call the `createLine()` method of the `SalesLine` table to create a new line. This is a very useful method, which allows you to quickly create sales order lines. The method accepts a number of optional Boolean arguments. The following list explains most of them:

- Perform the data validations before saving; the default is `false`
- Initialize the line record from the `SalesTable` table; the default is `false`
- Initialize the line record from the `InventTable` table; the default is `false`
- Calculate the inventory quantity; the default is `false`

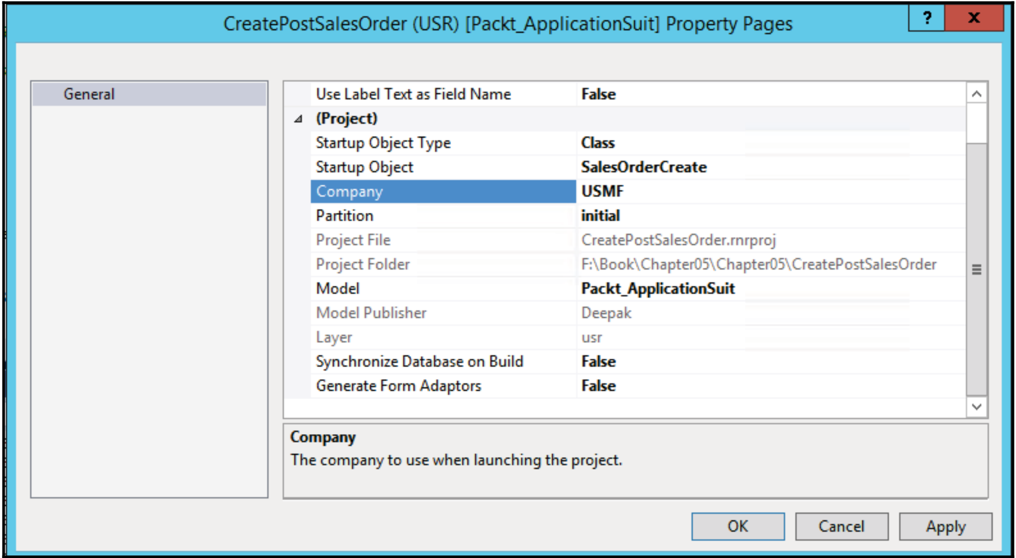
- Add the miscellaneous charges; the default is `true`
- Use the trade agreements to calculate the item price; the default is `false`
- Reserve the item; the default is `false`
- Ignore the customer credit limit; the default is `false`

There's more...

You can also use the data entities to insert **Sales Order Header** and **Line records**. To insert into Sales order header, use the `SalesOrderHeaderEntity` entity, and for Sales Order Line data, use the `SalesOrderLineEntity` data entity.

In the preceding code sample, we used a few methods to set some mandatory values in `SalesTable` and `SalesLine`. You won't find similar methods in these entities so you may have to assign all mandatory values manually.

While running any code from VS directly, it uses application UI to perform this task. In this situation, many times you don't have the option to choose company and the default company will be DAT. So you have to set the company on your project before you run your code. To set the default company for a specific project, set the project property as shown in the following screenshot:



Posting a sales order

In Dynamics 365 for Finance and Operations, a sales order goes through a number of statuses in order to reflect its current position within the sales process. The status can be updated either manually using the user interface or programmatically from the code.

In this recipe, we will demonstrate how a sales order status can be updated from the code. We will register a packing slip for the sales order created in the previous recipe and print the relevant document on the screen.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Add a new `runnable class`, named `SalesOrderPostPackingSlip` with the following code snippet in the main method (replace `000776` with your Sales Order number, that was generated after the previous code):

```
 static void SalesOrderPostPackingSlip(Args _args)
{
   SalesFormLetter salesFormLetter;
   salesTable      salesTable;
   salesTable = SalesTable::find('000776');
   salesFormLetter = SalesFormLetter::construct(
    DocumentStatus::PackingSlip);
   salesFormLetter.update(
    salesTable,
   DateTimeUtil::date(DateTimeUtil::utcNow()),
   SalesUpdate::All,
   AccountOrder::None,
   NoYes::No,
   NoYes::Yes);
}
```

2. Save and build your code and select this class as **set as startup object**. Now, run the project to post the specified sales order. As a result you will see the status of sales order `000776` will be changed to delivered.

How it works...

In this recipe, we create a new job named SalesOrderPostPackingSlip, which holds all the code.

First, we find a sales order, which we are going to update. In this recipe, we use the sales order created in the previous recipe. Here, we will normally replace this code with a user input or an output from some other function.

Next, we create a new SalesFormLetter object using its construct() constructor. The constructor accepts an argument of the DocumentStatus type, which defines the type of posting to be done. Here, we use DocumentStatus::PackingSlip as a value, as we want to register a packing slip.

Finally, we call the update() method of SalesFormLetter, which does the actual posting. It accepts a number of arguments, as follows:

- The sales order header record, that is, the SalesTable table.
- The transaction date; the default is the system date.
- The quantity to be posted; the default is SalesUpdate::All.
- The order summary update; this argument is not used at all. The default is AccountOrder::None.
- A Boolean value defining whether a preview or the actual posting should be done.
- A Boolean value defining whether the document should be printed.
- A Boolean value specifying whether printing management should be used; the default is false.
- A Boolean value defining whether to keep the remaining quantity on order; otherwise, it is set to zero. This argument is used when posting credit notes.
- A container of a number of TmpFrmVirtual records; this argument is optional and is used only when posting sales invoices.

There's more...

The SalesFormLetter class can also be used to do other types of posting, such as sales order confirmation, picking lists, or invoices. For example, to invoice the previously used sales order:

```
salesFormLetter = SalesFormLetter::construct(
 DocumentStatus::PackingSlip);
```

Replace the preceding line of code with the following line of code:

```
salesFormLetter = SalesFormLetter::construct(
 DocumentStatus::Invoice);
```

Now, when you run the job, the sales order will be updated to an invoice.

Creating an electronic payment format

Electronic payments, in general, can save time and reduce paperwork when making or receiving payments within a company. Dynamics 365 for Finance and Operations provides a number of standard out-of-the-box electronic payment formats. The system also provides an easy way of customizing the existing payment forms or creating new ones.

In this recipe, you will learn how to create a new custom electronic payment format. To demonstrate the principle, we will only output some basic information, and we will concentrate on the approach itself.

How to do it...

Carry out the following steps in order to complete this recipe:

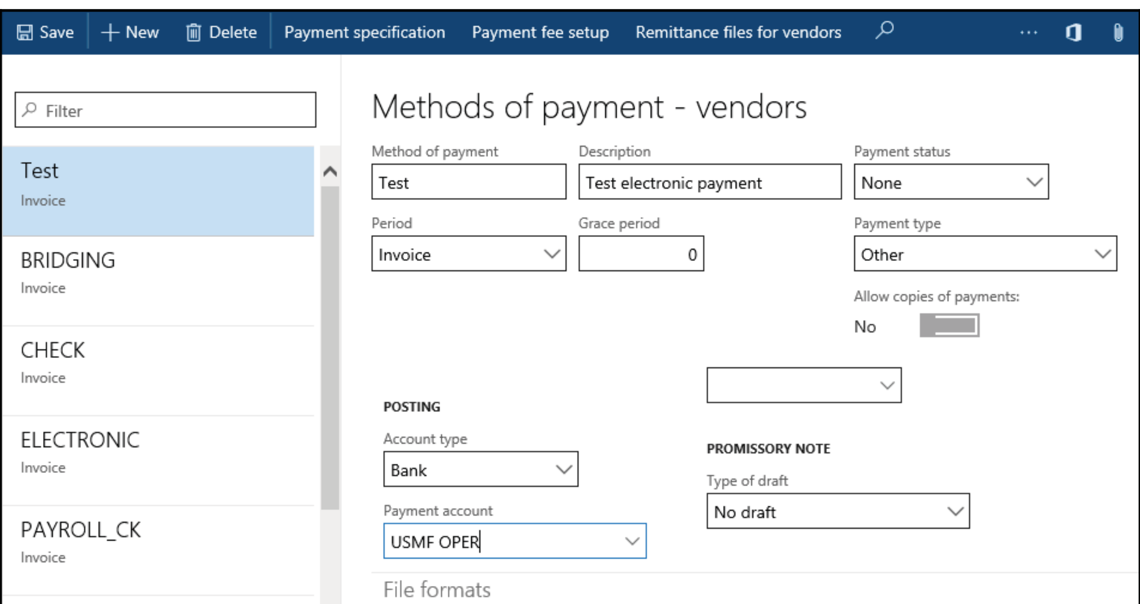
1. In the AOT, create a new class named `VendOutPaymRecord_Test` with the following code snippet:

```
 public class VendOutPaymRecord_Test extends VendOutPaymRecord
{
}
 public void output()
{
   str         outRecord;
   Name        companyName;
   BankAccount bankAccount;
   outRecord = strRep(' ', 50);
   companyName = subStr(
   custVendPaym.recieversCompanyName(), 1, 40);
   bankAccount = subStr(
   custVendPaym.recieversBankAccount(), 1, 8);
   outRecord = strPoke(outRecord, companyName, 1);
   outRecord = strPoke(outRecord, bankAccount, 43);
   file.write(outRecord);
}
```

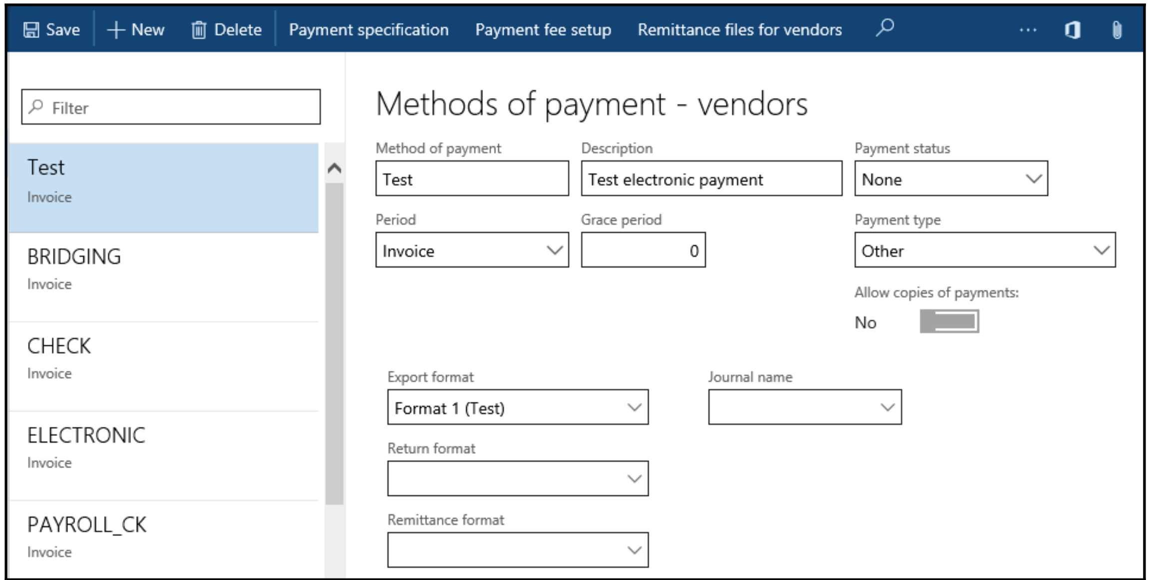
2. Create another class named `VendOutPaym_Test` with the following code snippet:

```
public class VendOutPaym_Test extends VendOutPaym
{
}
public PaymInterfaceName interfaceName()
{
   return "Test payment format";
}
public ClassId custVendOutPaymRecordRootClassId()
{
   return classNum(VendOutPaymRecord_Test);
}
protected Object dialog()
{
   DialogRunbase dialog;
   dialog = super();
   this.dialogAddFileName(dialog);
   return dialog;
}
public boolean validate(Object _calledFrom = null)
{
   return true;
}
public void open()
{
   #LocalCodePage
   file = CustVendOutPaym::newFile(filename, #cp_1252);
   if (!file || file.status() != IO_Status::Ok)
    {
       throw error(
        strFmt("File %1 could not be opened.", filename));
    }
    file.outFieldDelimiter('');
    file.outRecordDelimiter('\r\n');
    file.write('Starting file:');
}
public void close()
{
   file.write('Closing file');
}
```

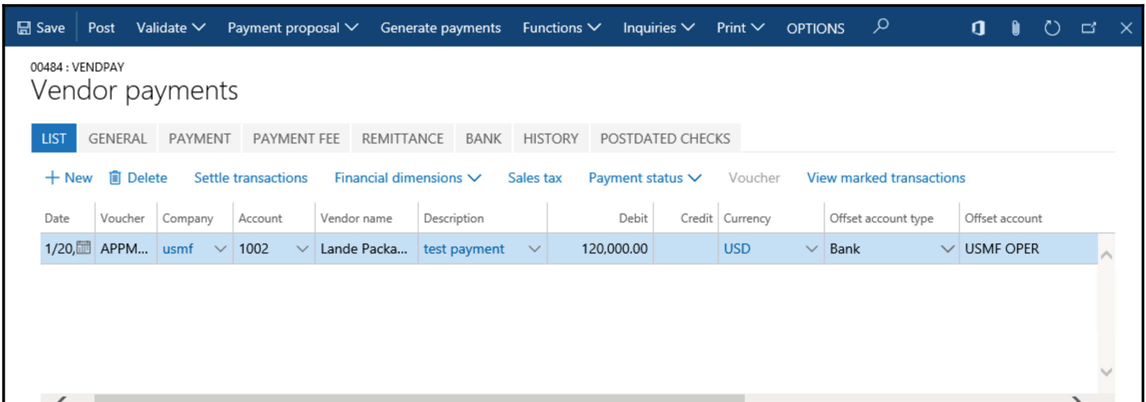
3. Navigate to **Accounts payable** | **Setup** | **Payment** | **Methods of payment** and create a new record, as follows:



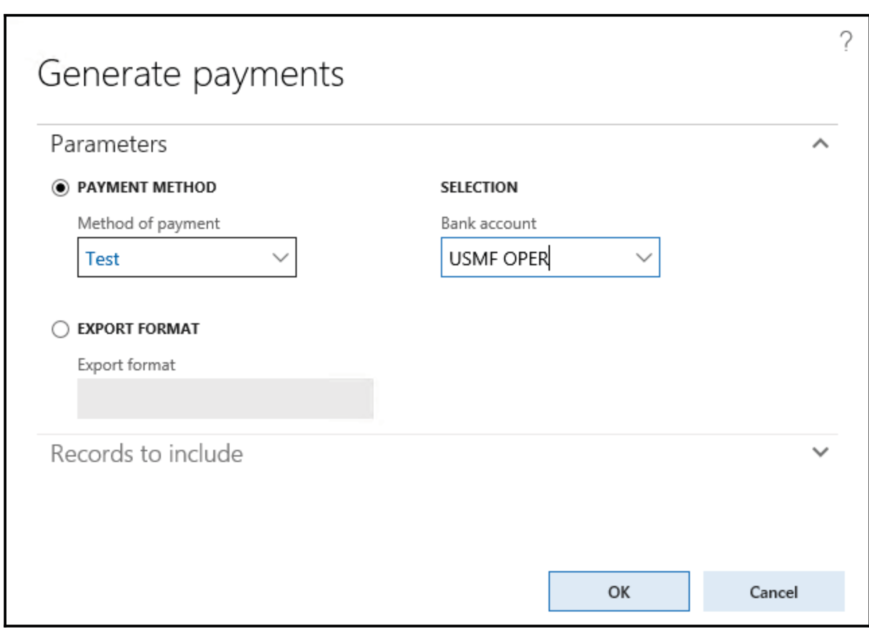
4. Open the **File formats** tab page, click on the **Setup** button, and move your newly created **Test payment format** from the pane on the right-hand side to the pane on the left-hand side.
5. Then, go back to the **Methods of payment** form and select **Text payment format** in the **Export format** field as follows:



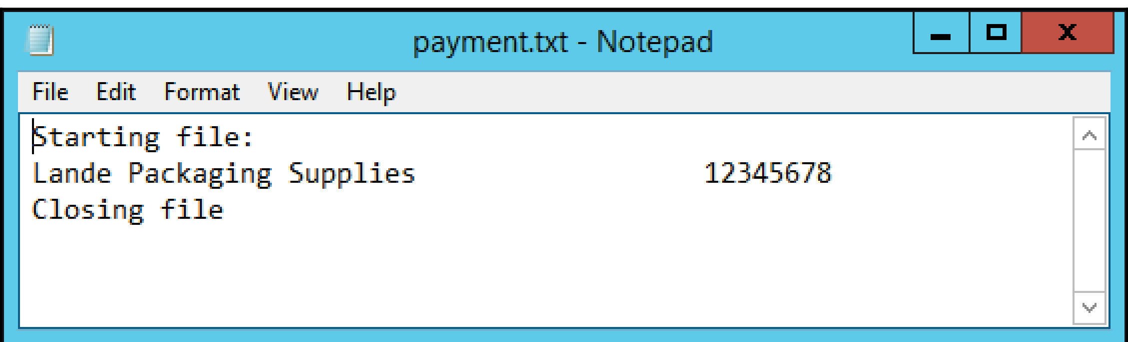
6. Close the **Methods of payment** form. Navigate to **Accounts payable** | **Journals** | **Payments** | **Payment journal** and create a new journal. Click on the **Lines** button to open the journal lines. Create a new line and make sure you set **Method of payment** to **Test**, as follows:



7. Next, click **Generate payments**. Fill in the dialog fields as displayed in the following screenshot, click on **OK**, and select the exported file's name:



8. Click on **OK** to complete the process; notice that the journal line's **Payment status** changed from **None** to **Sent**, which means that the payment file was generated successfully.
9. Open the created file with any text editor (for example, Notepad), to check its contents, shown as follows:



How it works...

In this recipe, we create two new classes, which are normally required for generating custom vendor payments. Electronic payments are presented as text files to be sent to the bank. The first class is the `VendOutPaymRecord_Test` class, which is responsible for formatting the payment lines, and the second one is the `VendOutPaym_Test` class, which generates the header and footer sections and creates the payment file itself.

The `VendOutPaymRecord_Test` class extends `VendOutPaymRecord` and inherits all the common functionality. We only need to override its `output()` method to define our own logic in order to format the payment lines. The `output()` method is called once for each payment line.

Inside the `output()` method, we use the `outRecord` variable, which we initially fill in with **50** blank characters using the global `strRep()` function, and then insert all the necessary information into the predefined positions within the variable, as per format requirements. Normally, here we should insert all the required information, such as dates, account numbers, amounts, references, and so on. However, to keep this demonstration to a minimum, we only insert the company name and the bank account number.

In the same method, we use another variable named `custVendPaym` of the `CustVendPaym` type, which already holds all the information we need. We only have to call some of its methods to retrieve it. In this example, to get the company name and the bank account number, we call `recieversCompanyName()` and `recieversBankAccount()`, respectively. We trim the returned values using the global `substr()` function, and insert them into the **first** and **43rd positions** of the `outRecord` variable using the global `strPoke()` function.

Finally, at the bottom of the `output()` method, we add the formatted text to the end of the payment file.

Another class that we create is `VendOutPaym_Test`. It extends the `VendOutPaym` class and also inherits all the common functionality. We only need to override some of the methods that are specific to our format.

The `interfaceName()` method, returns a name of the payment format. Normally, this text is displayed in the user interface, when configuring payments.

The `custVendOutPaymRecordRootClassId()` method returns an ID of the class, which generates payment lines. It is used internally to identify which class to use when formatting the lines. In our case, it is `VendOutPaymRecord_Test`.

The `dialog()` method is used only if we need to add something to the user screen when generating payments. Our payment is a text file, so we need to ask a user to specify the filename. We do this by calling the `dialogAddFileName()` method, which is a member method of the parent class. It will automatically add a file selection control and we won't have to worry about things, such as a label or how to get its value from the user input. There are numerous other standard controls, which can be added to the dialog by calling various `dialogAdd...()` methods. Additional controls can also be added here using `addField()` or similar methods of the dialog object directly.

The `validate()` method is one of the methods that has to be implemented in each custom class. Normally, user input validation should go here. Our example does not have any validation, so we simply return true.

In the `open()` method, we are responsible for initializing the file variable for further processing. Here, we use the `newFile()` constructor of the `CustVendOutPaym` class to create a new instance of the variable. After some standard validations, we set the field and the row delimiters by calling the `outFieldDelimiter()` and `outRecordDelimiter()` methods of the `CustVendOutPaym` class, respectively. In this example, the values in each line should not be separated by any symbol, so we call the `outFieldDelimiter()` method with an empty string. We call the `outRecordDelimiter()` method with the new line symbol to define that every line ends with a line break. Note that the last line of this method writes a text to the file header. Here, we place some simple text so that we can recognize it later when viewing the generated file.

The last one is the `close()` method, which is used to perform additional actions before the file is closed. Here, we specify some text to be displayed in the footer of the generated file.

Now, this new payment format is ready for use. After some setup, we can start creating the vendor payment journals with this type of payment. Note, the file generated in the previous section of this recipe, we can clearly see which text in the file comes from which part of the code. These parts should be replaced with your own code to build custom electronic payment formats for Dynamics 365 for Finance and Operations.

6
Data Management

In this chapter, we will cover the following recipes:

- Data entities
- Building a data entity with multiple data sources
- Data packages
- Data migration
- Import of data
- Troubleshooting

Introduction

The data management feature in Dynamics 365 for Finance and Operations enables you to manage and audit your data efficiently in systems. The excellent feature provides many tools such as Import, Export, delete bulk data and detect duplicate data, and so on. You can also develop custom data entities as well.

Integration through the data management platform provides more capabilities and higher throughput for inserting/extracting data through entities. Typically, data goes through three phases in this integration scenario:

- **Source** - These are inbound data files or messages in the queue. Typical data formats include CSV, XML, and tab-delimited.
- **Staging** - These are automatically generated tables that map very closely to the data entity. When data management enabled is true, staging tables are generated to provide intermediary storage. This enables the framework to do high-volume file parsing, transformation, and some validations.
- **Target** - This is the data entity where data will be imported.

Now let's see how to build an entity and how to use any existing/new data entity in Dynamics 365 for Finance and Operations. We can create new entities in two ways:

- Using a Wizard
- Directly from a table

We will explain both with different recipes in this chapter.

Data entities

In the earlier version of Dynamics 365 for Finance and Operations, there are multiple options such as DIXF, Excel Add-ins, and AIF for data management. Data entities are introduced as a part of data management to be used as a layer of abstraction to easily understand by using business concepts.

The concept of data entities combines those different concepts into one. You can reuse Data entities for an Excel Add-ins, Integration, or import/export. The following table shows core scenarios of Data management:

Data Migration	Migrate reference, master, and document data from legacy or external systems.
Setup and copy configuration	Copy configuration between company/environments. Configure processes or modules using the Lifecycle Services (LCS) environment.
Integration	Real-time service based integration. Asynchronous integration.

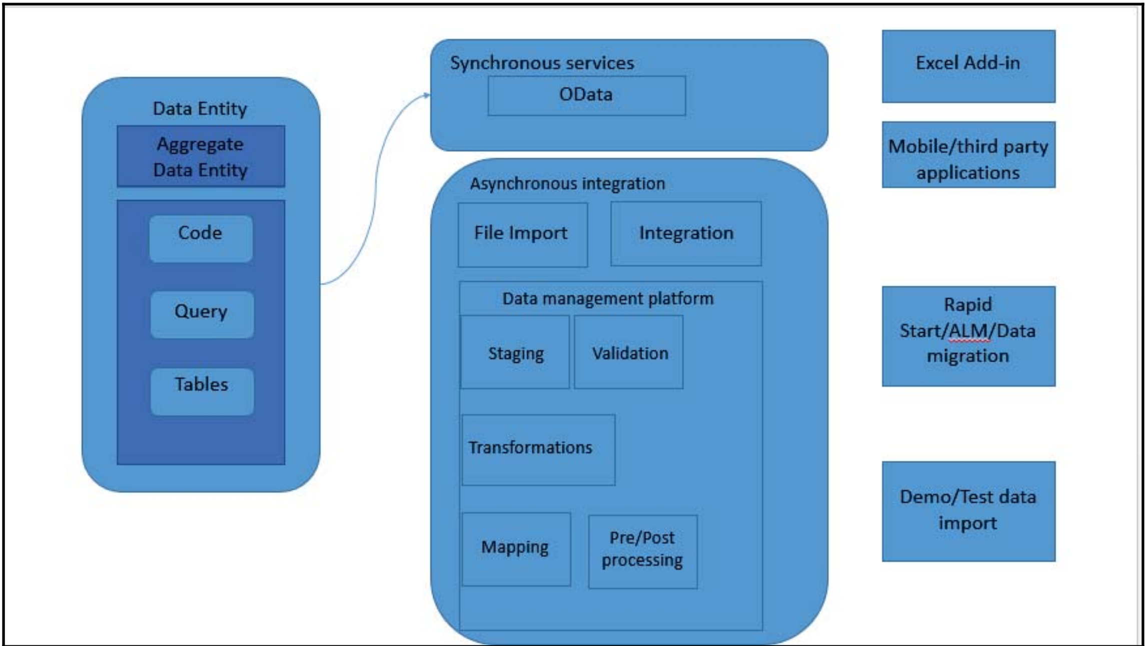
More information about this can be found at `https://docs.microsoft.com/en-us/dynamics365/unified-operations/dev-itpro/data-entities/data-entities-data-packages`.

Getting ready

The following are the terms introduced for data management that will be used throughout the chapter:

Data project	A project that contains configured data entities, which include mapping and default processing options.
Data job	A job that contains an execution instance of the data project, uploaded files, schedule (recurrence), and processing options.
Job history	Histories of source to staging and staging to target.
Data package	A single compressed file that contains a data project manifest and/or data files. This is generated from a data job and used for import or export of multiple files with the manifest.

Data management uses data entities under the hood for an abstract layer for business logic implementation. Data is inserted in staging tables using SSIS, which is then validated and transformed to map to the target entity.

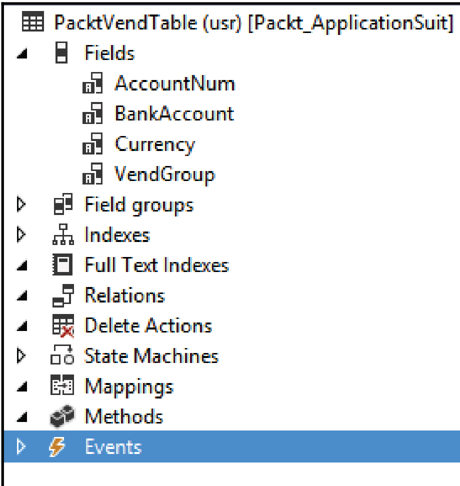


How to do it...

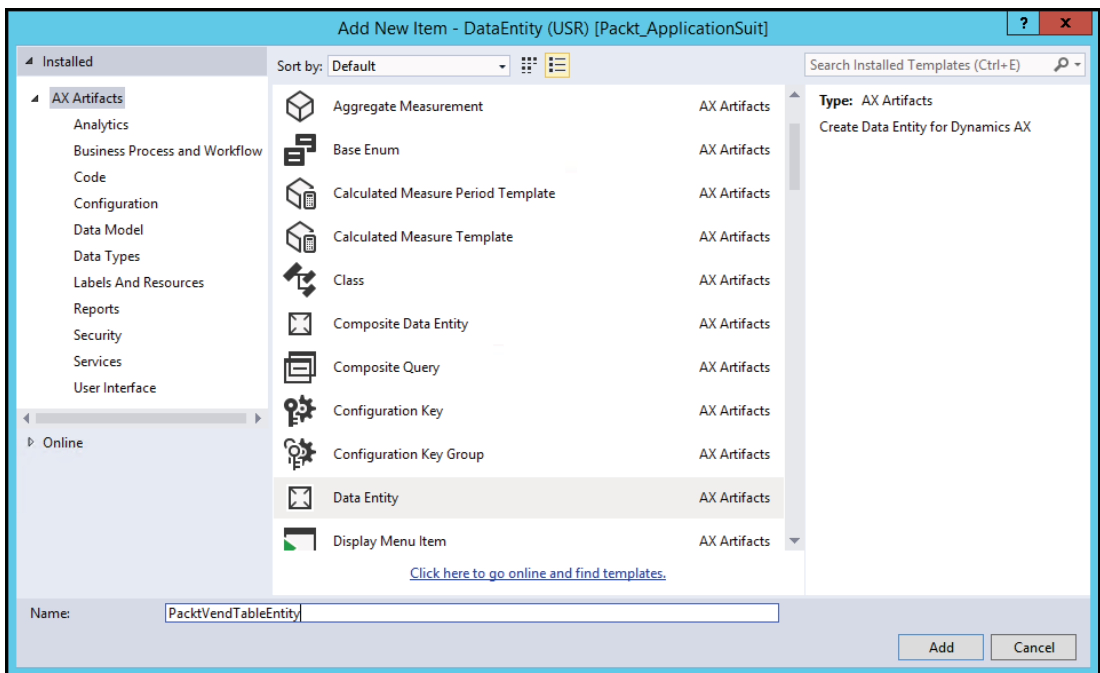
Carry out the following steps in order to complete this recipe:

1. Create a new Dynamics 365 for Operations project in Visual Studio.

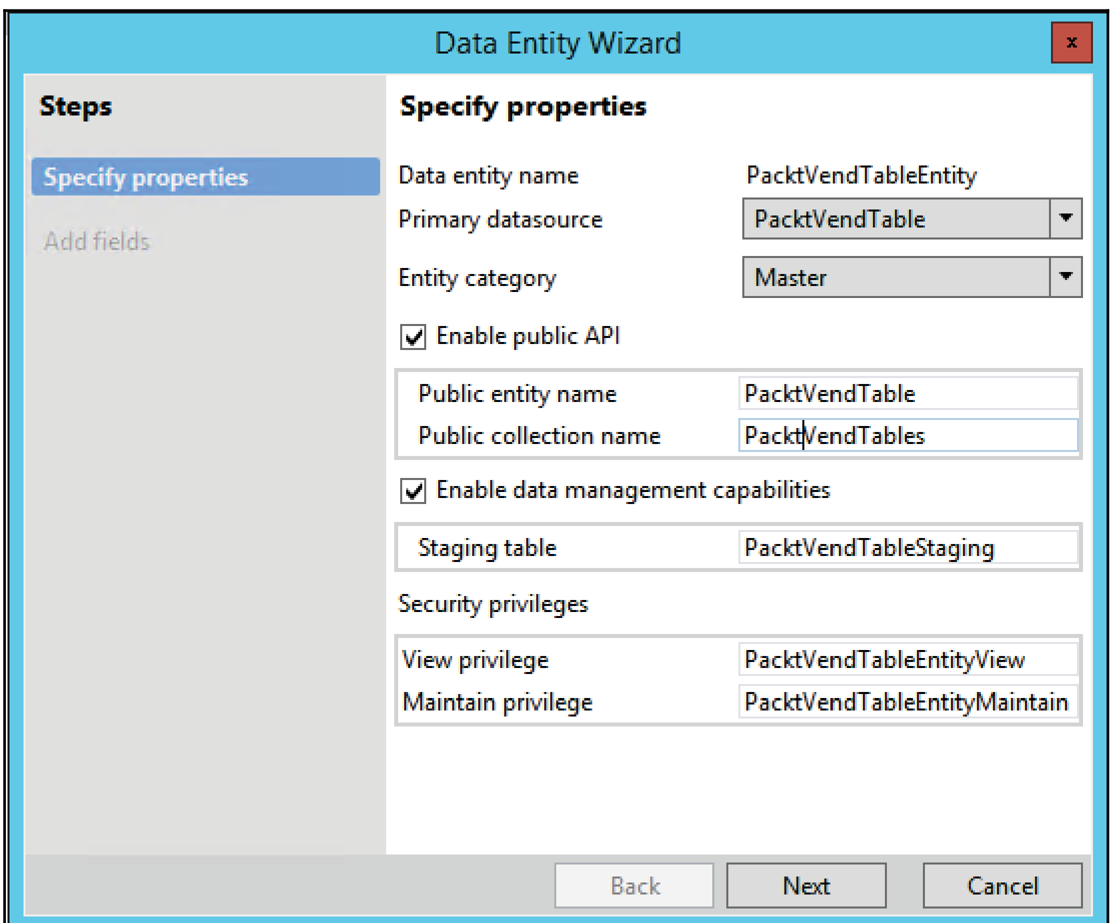
2. We will create a demo table with a few fields as follows, to use in this **Data Entity**:



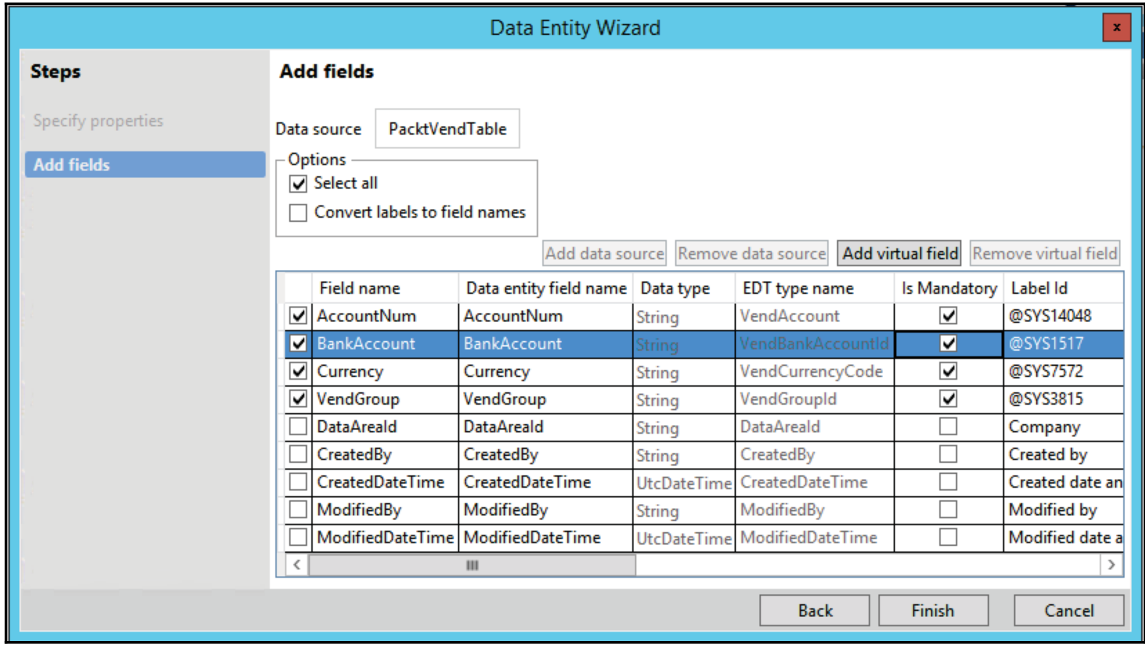
3. Add a new **Data Entity** in the project by right-clicking the menu as follows:



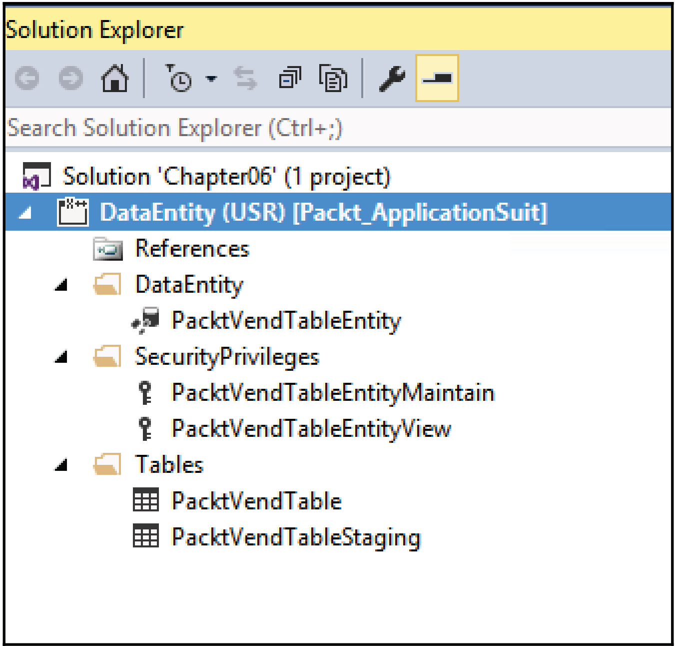
4. Next you will get a wizard screen, select **PacktVendTable** as the **Primary datasource**. **Entity category** as **Master** and click on **Next**:



5. In the next step, you have to choose all required fields from the primary `dataSource` table, for this recipe we will keep only a few fields and mark them as **Is Mandatory** as well. Once done, click on **Finish**:

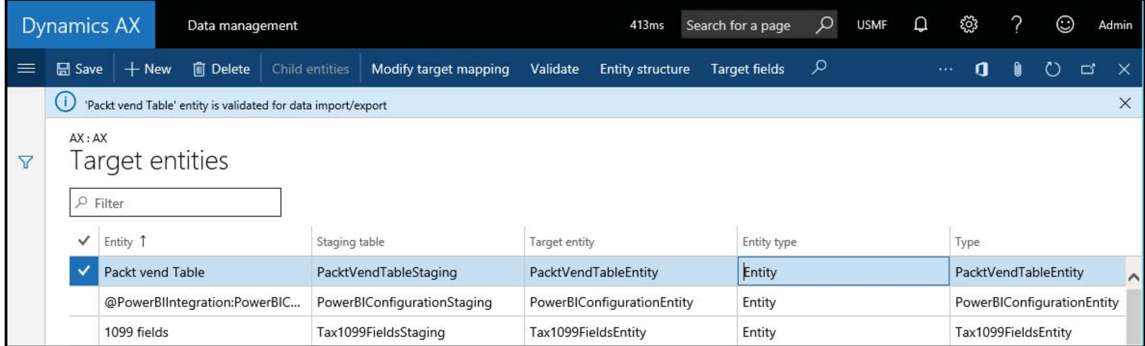


6. Save your project and build it. The project must look as follows:

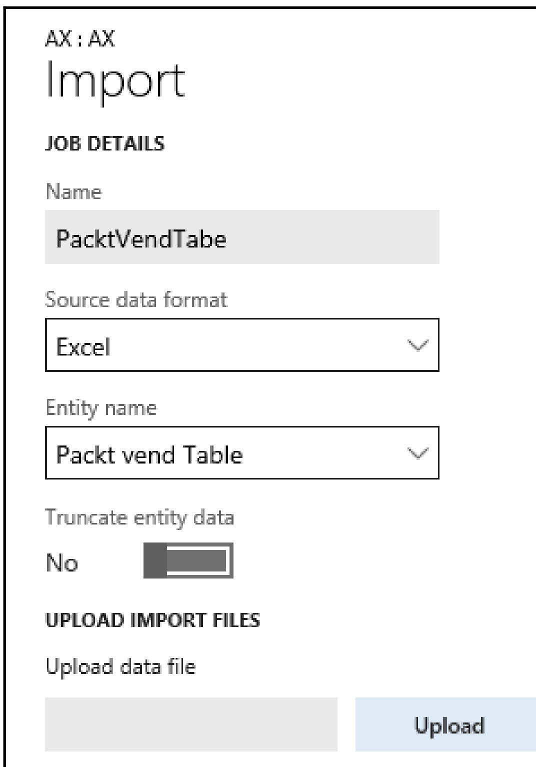


7. Now we need to add this entity in the data management work space. Navigate to **Work space** | **Data management**|**Data entities.**

   Add a new record as follows and save it. Now click on **Validate**:



8. Let's try to import data into **PacktVendTable** using this new Data entity. Now go back to the **Data management** work space. Click on **Import tile**. Fill in the details as follows:



Upload the Excel file that contains data. Now click on the **Import** button. You will get a notification once this import is done. To check, browse the table and check inserted data.

How it works...

We start this recipe with creating a new table with a few fields similar to `vendTable`. This table is used to create a new data entity through the VS wizard. We added new a data entity object in our project, once we select a Data entity object it will initiate a wizard.

In this step, we select our primary table that we created earlier. In the entity category field you have to choose this on the basis of table type. There are five different types of entity category. If you are using any existing table, this will select automatically, while for new tables you may have to change it accordingly.

In the next step, we select fields that are really required in this entity. You can change a few properties of fields such as label and mandatory. At the end of this wizard you will have a new data entity along with a staging table created. You will find a few more supporting Dynamics 365 for Finance and Operations objects in your project.

There's more...

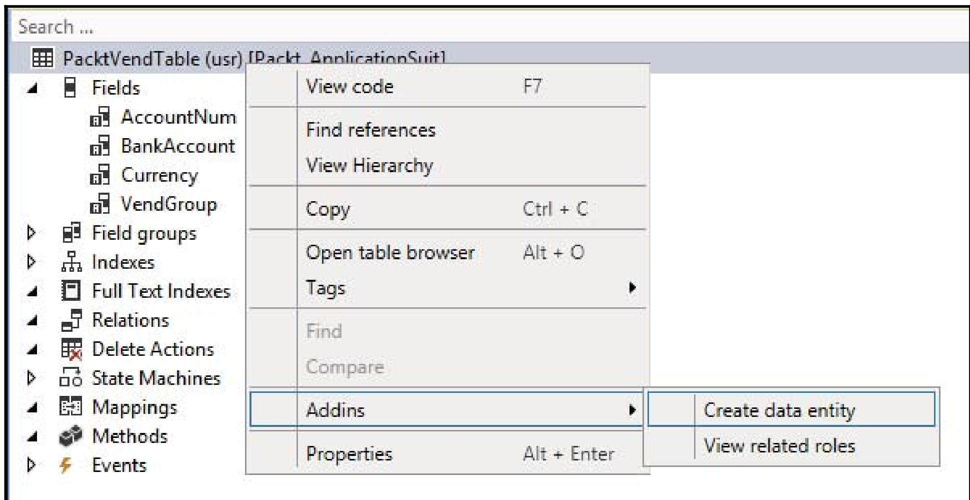
It is important to understand the different categories of entities while you are working on data entities. In Dynamics 365 for Finance and Operations, entities are categorized based on their functions and the type of data that they serve. The following are five categories for data entities:

- **Parameter:**
    - Tables that contain only one record, where the columns are values for settings. Examples of such tables exist for Account payable (AP), General ledger (GL), client performance options, workflows, and so on.
    - Functional or behavioral parameters.
    - Required to set up a deployment or a module for a specific build or customer.
    - Can include data that is specific to an industry or business. The data can also apply to a broader set of customers.
- **Reference:**
    - Simple reference data, of small quantity, that is required to operate a business process.
    - Data that is specific to an industry or a business process.
    - Examples include units, dimensions, and tax codes.
- **Master:**
    - Data assets of the business. Generally, these are the "nouns" of the business, which typically fall into categories such as people, places, and concepts.
    - Complex reference data, of large quantity. Examples include customers, vendors, and projects.

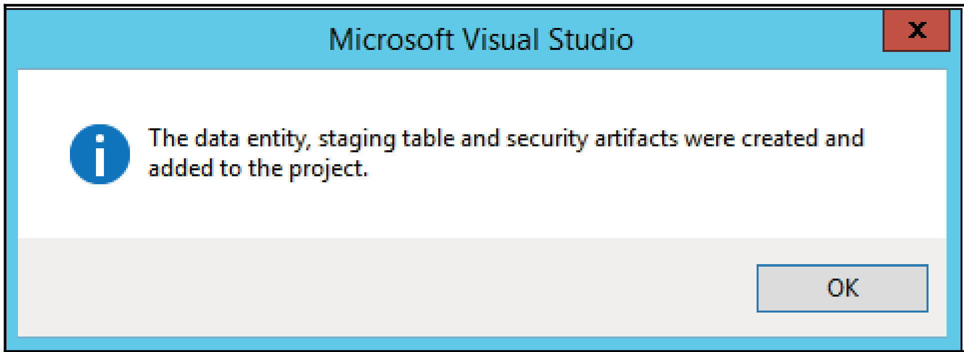
- **Document:**
  - Worksheet data that is converted into transactions later.
  - Documents that have complex structures, such as several line items for each header record. Examples include sales orders, purchase orders, open balances,and journals.
  - The operational data of the business.
- **Transaction:**
  - The operational transaction data of the business.
  - Posted transactions. These are non-idempotent items such as posted invoices and balances. Typically, these items are excluded during a full dataset copy.
  - Examples include pending invoices.

Let's see one more example where we will discuss how to create the same entity from the **PacktVendTable** wizard. To carry on, follow these steps:

1. Right-click on the table and select **Addins** | **Create data entity**. As shown in the following screenshot:



2. It will directly create all required objects in your current project:



3. Now save all your changes and **build** the solution. On successful build, your Data entity will be ready for use.

Building a data entity with multiple data sources

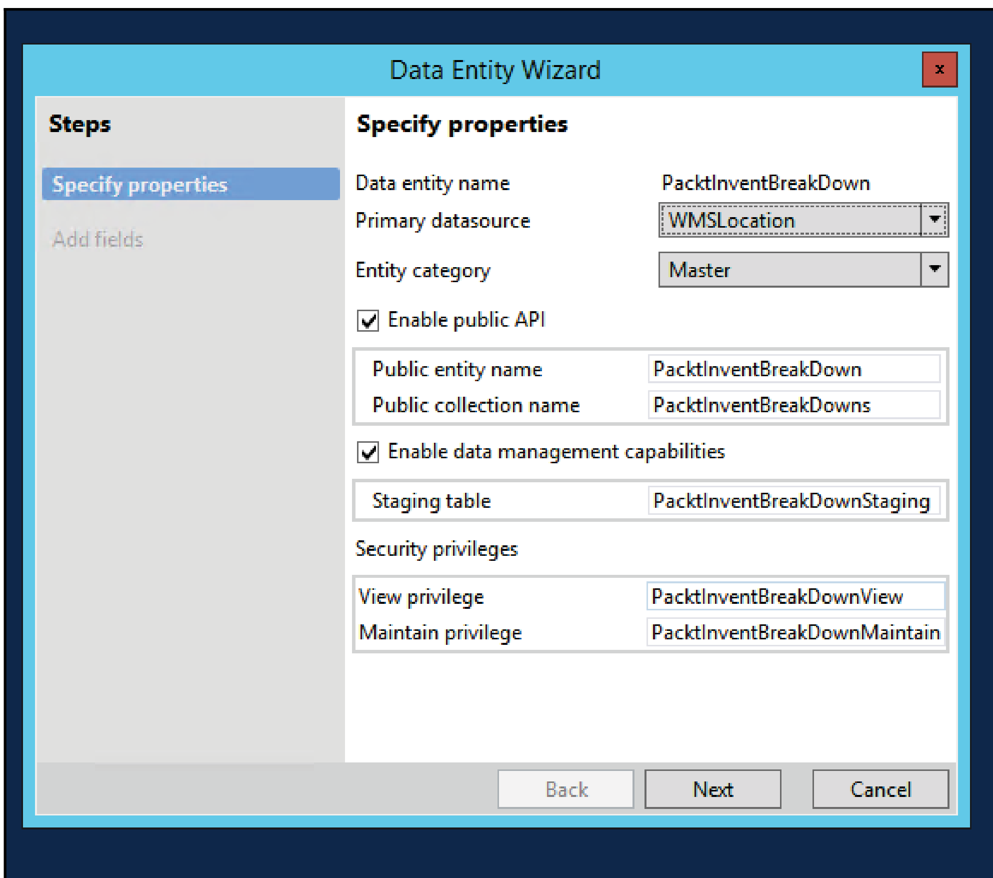
We could also create a data entity where we include multiple data sources. Here our data entity takes care of all integrity constraints and validation and creates records in related tables if it does not exist. Let us take, an example of inventory breakdown, where we create an inventory site, warehouse, location, zones, aisle, and so on. We could create a data entity, which encapsulates all these tables, and a flat file import could create related records in all these tables.

How to do it...

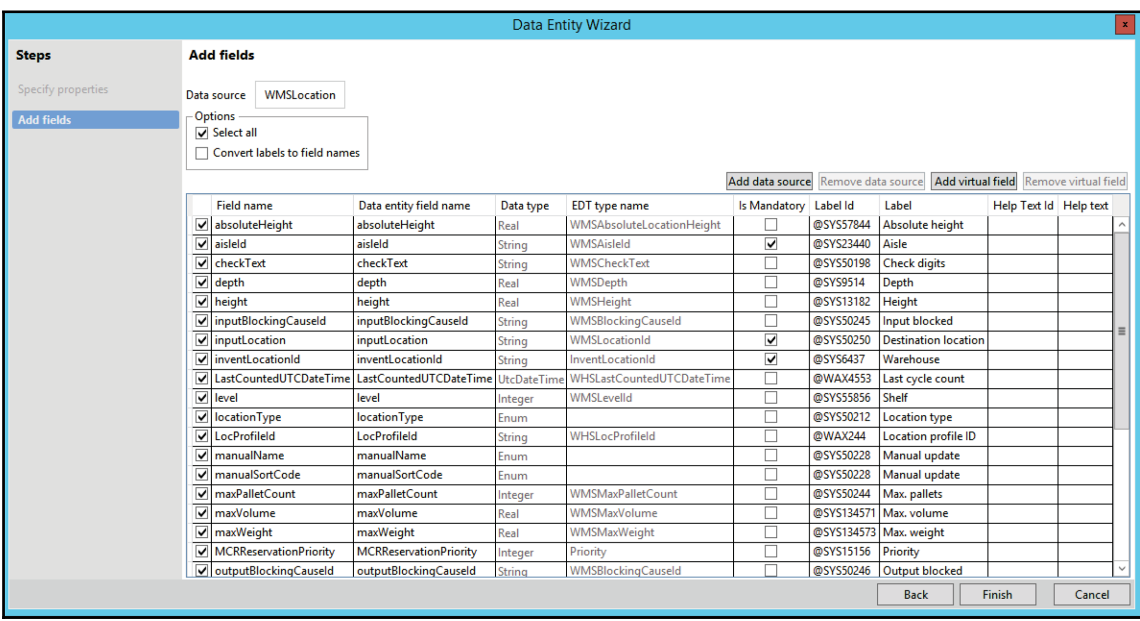
Carry out the following steps in order to complete this recipe:

1. Add a new data entity in the project and name it `PacktInventBreakDown`.

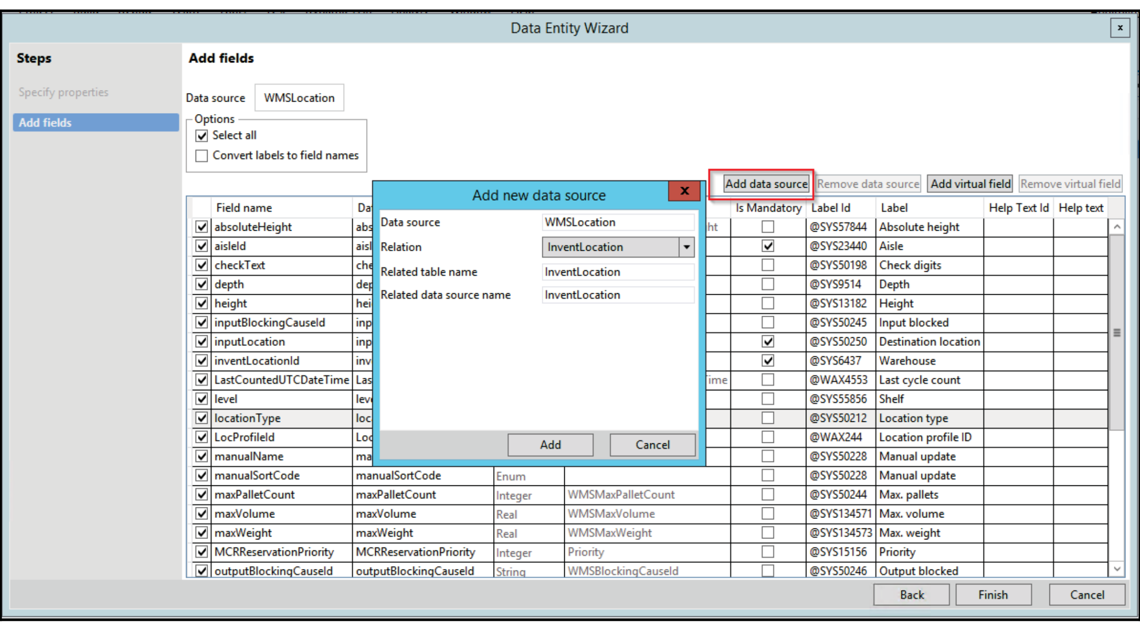
2. A **Data Entity Wizard** will be launched, as shown in the following screenshot:



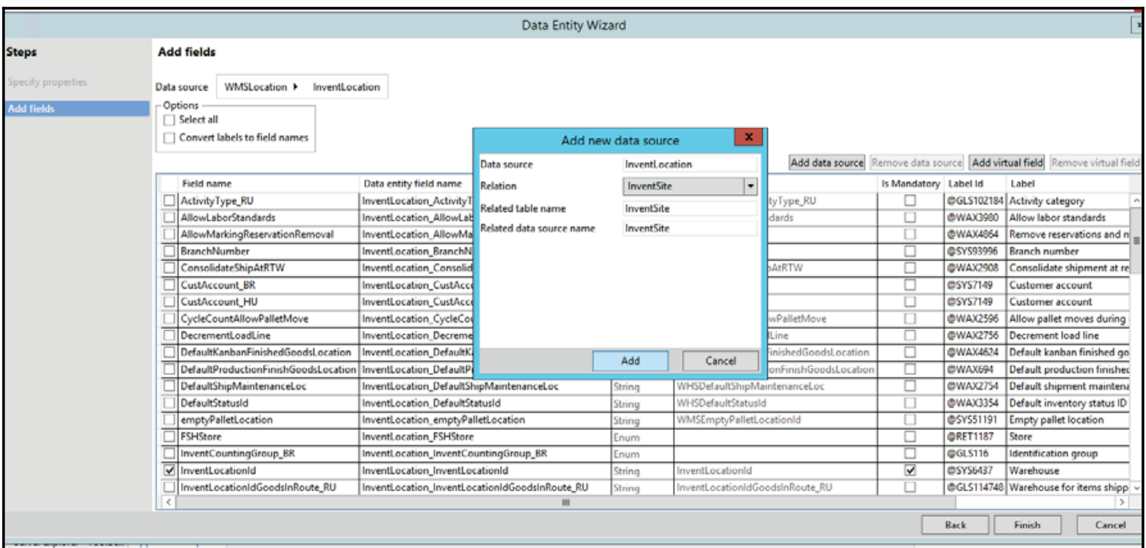
3. Next you need to select **all/required fields** from **WMSLocation**.



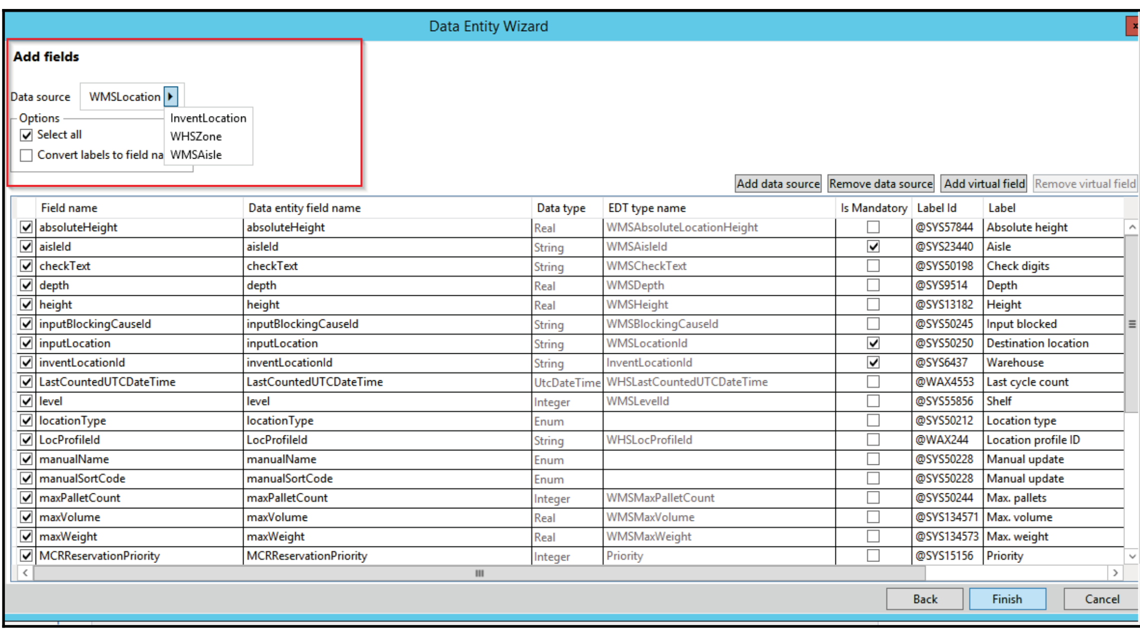
4. Click on the **Add data source** button and select **Relation InventLocation**.



5. Select **Invent location** from the node at the right of **WMSlocation,** as shown in the next screenshot.
6. **Add new data source InventSite** to **InventLocation** and select all fields:



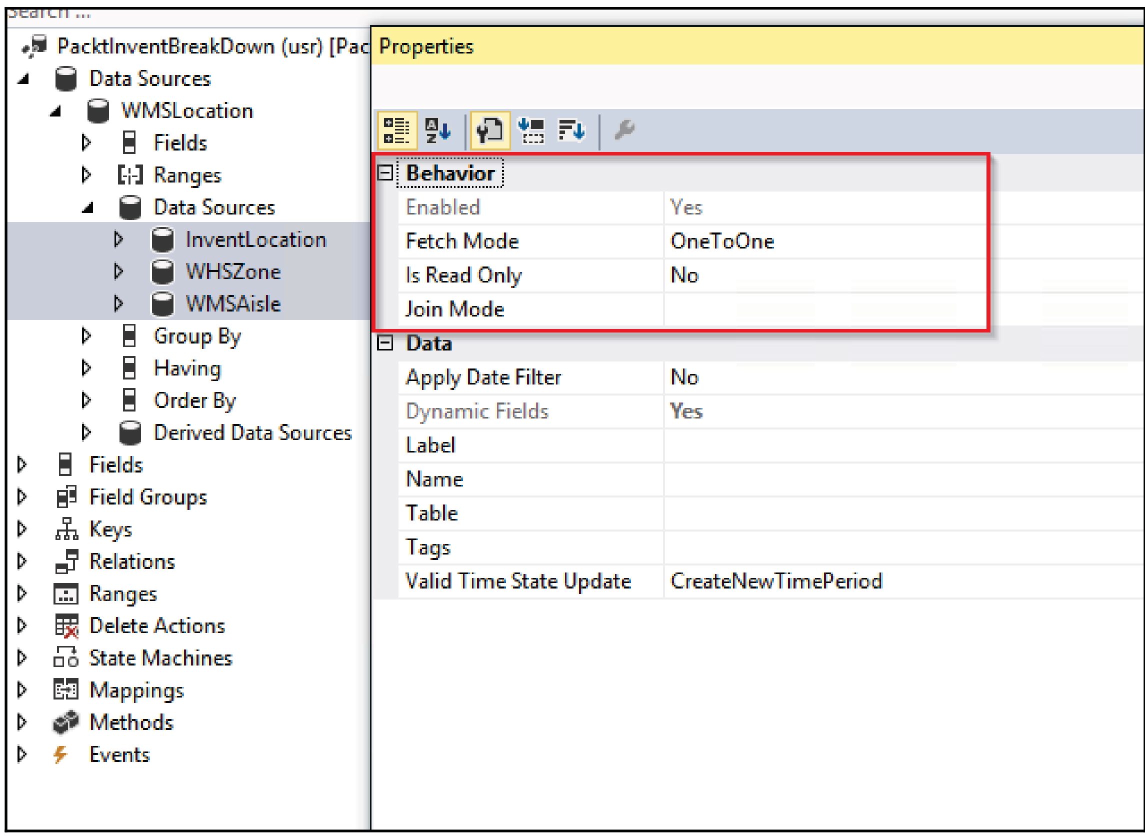
7. Add more tables, **WHSZone** and **WMSAisle,** on **WMSLocation**:



8. The system will create the data entity, staging table, and privileges to support data management and OData on the data entity.
9. Select all child data sources and set the `Is Read Only` property as **No,** as shown in the following table:

Your properties must look as follows:

Fetch mode	OneToOne
Is Read Only	No



10. Now to verify the integrity of the data entity. Create a runnable class name, `InventBreakDownCreate`, and add the following code:

```
class InventBreakDownCreate
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
 {
   PacktInventBreakDown inventBreakDown;

   inventBreakDown.initValue();
   inventBreakDown.InventSite_SiteId    = 'PacktSite';
   inventBreakDown.InventSite_Name = "Packt site":
   inventBreakDown.InventLocation_InventLocationId = "Packt11";
   inventBreakDown.InventLocation_Name = "Packt 11";
```

```
        inventBreakDown.wMSLocationId = "PacktWMS11";
        inventBreakDown.aisleId = "PacktWMSAisle";
        inventBreakDown.WMSAisle_inventLocationId = "Packt11";
        inventBreakDown.WMSAisle_aisleId ="PacktWMSAisle";
        inventBreakDown.WMSAisle_inventLocationId = "Packt11";
        inventBreakDown.inventLocationId = "Packt11";
        inventBreakDown.WHSZone_ZoneId = "PacktZone";
        inventBreakDown.WHSZone_ZoneName = "Packt Zone";
        inventBreakDown.WHSZone_ZoneGroupId = "BULK";
        inventBreakDown.ZoneId = "PacktZone";
        inventBreakDown.inputLocation = "PacktWMS11";
        inventBreakDown.locationType = WMSLocationType::Pick;
        inventBreakDown.insert();
      }
    }
```

11. On running this class it would create a site, warehouse, location, zone, and aisle in a single run. These types of data entities could be very useful when migrating flat data from a client.

How it works...

In this recipe, we used the **WMSLocation** table as our parent datasource. This table is used to create a new data entity through the VS wizard. We added a new data entity object in our project, once we select a data entity object it will initiate a wizard.

In this step, we select our primary table, that is, **WMSLocation**. In the entity category field, you have to choose this based on table type. There are five different types of entity category. If you are using any existing table, this will select automatically, while for new tables you may have to change it accordingly.

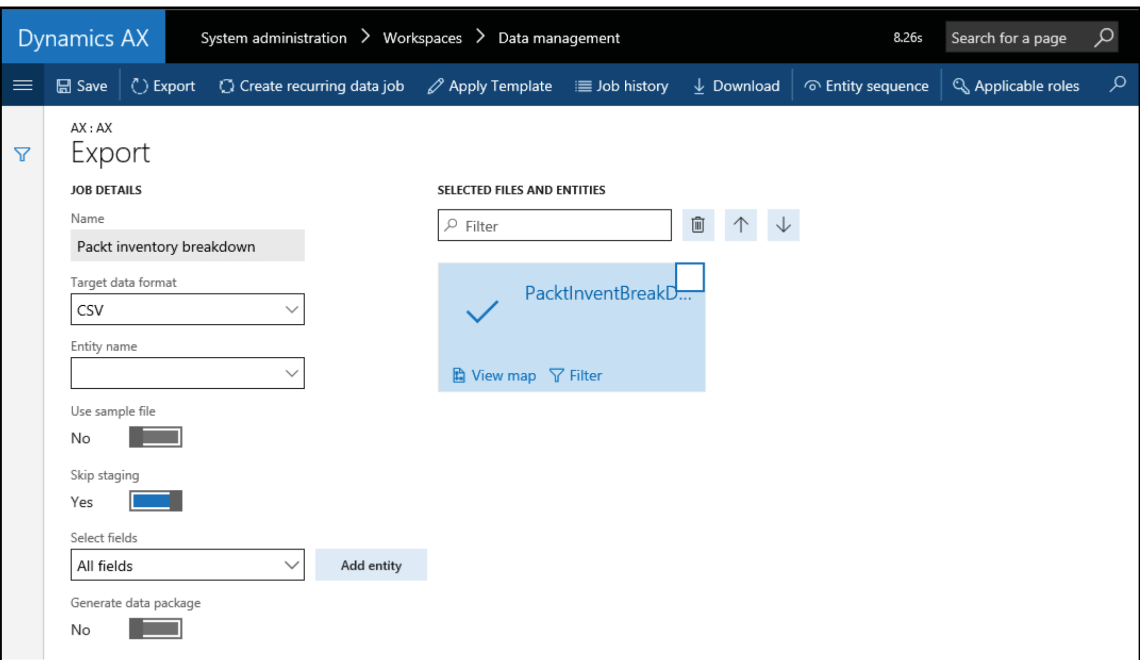
In the next step, we have selected **InventLocation**, **InventSite**, **WHSZone,** and **WMSAisle**, which are related tables. After you have found a few more supporting Dynamics 365 for Finance and Operations objects in your project. After the last step, you need to open data entity and modify the `Is Read Only` property on all child data sources to **No** as it would allow us to create related records in the child table. Finally, to test this entity you create a job and after running, verify the data in related tables.

You can add more data sources in the wizard and use fields from related child tables as well. To use other tables make sure relevant relations exist on table level.

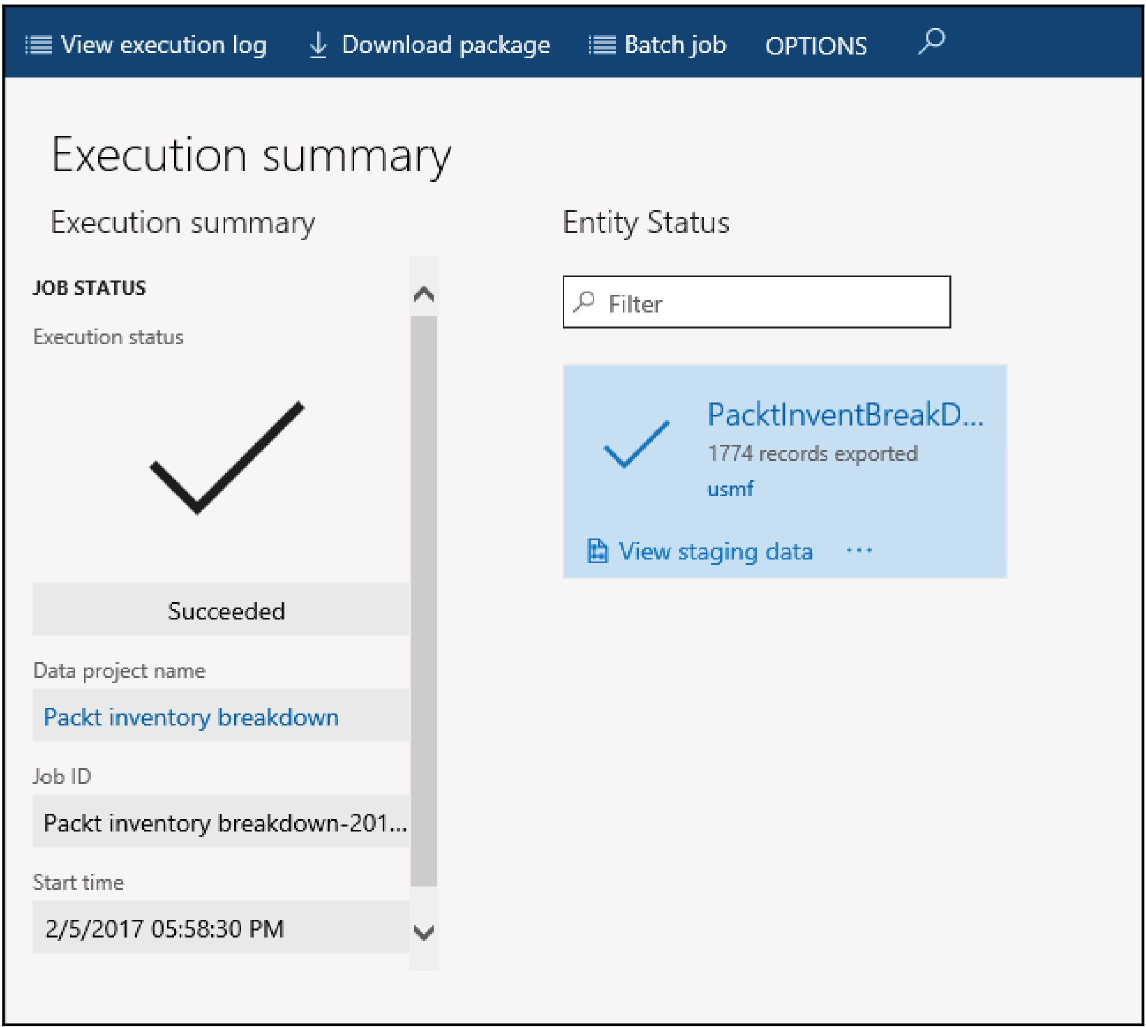
There's more...

The custom entity that we created in the preceding recipe could be used in data management. Here we would use the entity to export the inventory setup data.

Select the **Export** option and choose our custom entity in the Data entity drop-down, as shown in the following screenshot:



Select the **Export** option, it schedules an export batch job that reads the data entity and writes the data to file as per the setup specified for the export job. Data gets exported successfully and we can view staging data to validate that the correct data has been imported.



Data packages

Data packages in Dynamics consist of logically grouped data entities. In a simpler way, a data package contains one or more entities or groups of data entities. **Lifecycle Services (LCS)** contains multiple base data packages that you can use to reduce implementation time during the project. These packages also be used to prepare your system in much less time with demo/real data. These packages can contain the elements that are required in each module/area in order to meet the minimum requirements. As per business requirements or advanced business processes, you might have to add more entities to the list of packages.

Getting ready...

The data packages that Microsoft publishes on LCS use a numbering sequence that is based on the module, data type, and sequence. Here is an example:

- **Module/area numbering**:

Module	Module Reference
System administration	01
General ledger	03
Public Sector	04
HRM	05
Accounts payable	10
Accounts receivable	11
Budgeting	12
Cash and bank management	13
Compliance and internal controls	14
Cost accounting	15
Fixed assets	16
Inventory management	19
Master planning	20
Organization administration	21
Payroll	22
Procurement and sourcing	23
Product information management	24
Production control	25
Project management and accounting	26
Retail	27
Sales and marketing	28
Service management	29
Trade allowance management	31
Transportation management	32
Travel and expense	33
Warehouse management	34

- **Data type numbering**:

Data Type	Data Type Reference
Setup	1
Master	4
Transaction	8

- **Numbering format**:

Numbering Format	
Module # . Data Type Reference . 001 (Sequence Number)	
01.1.001	System / Setup Data / Seq 001
01.1.002	System / Setup Data / Seq 002
01.4.001	System / Master Data / Seq 001
01.4.002	System / Master Data / Seq 002
03.1.001	General Ledger / Setup Data / Seq 001

The names of data packages include the numbering format, which is followed by the module abbreviation and then a description. For example, the following screenshot shows the general ledger data packages:

03.1.001 GL - Exchange Rates
03.1.002 GL - Chart of Accounts
03.1.003 GL - Account Structures
03.1.004 GL - Fiscal Calendar
03.1.005 GL - Ledger Setup
03.1.006 GL - Ledger Journals
03.1.007 GL - Allocations

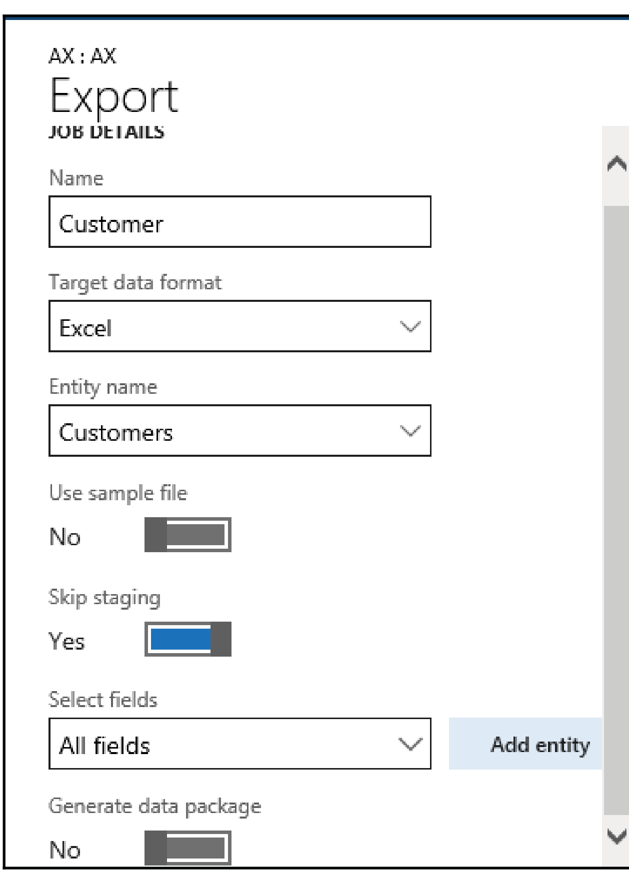
Now, in this recipe we will see how to create a new package in Dynamics 365 for Finance and Operations. However, it does not require any coding at all, but still you have to choose every entity wisely for your data package, to perform all export/import functions properly.

Data packages can be used for both importing and exporting data into your application. In coming recipes we will see both examples. Initially we will export a package followed by importing data back to the system using the same package.

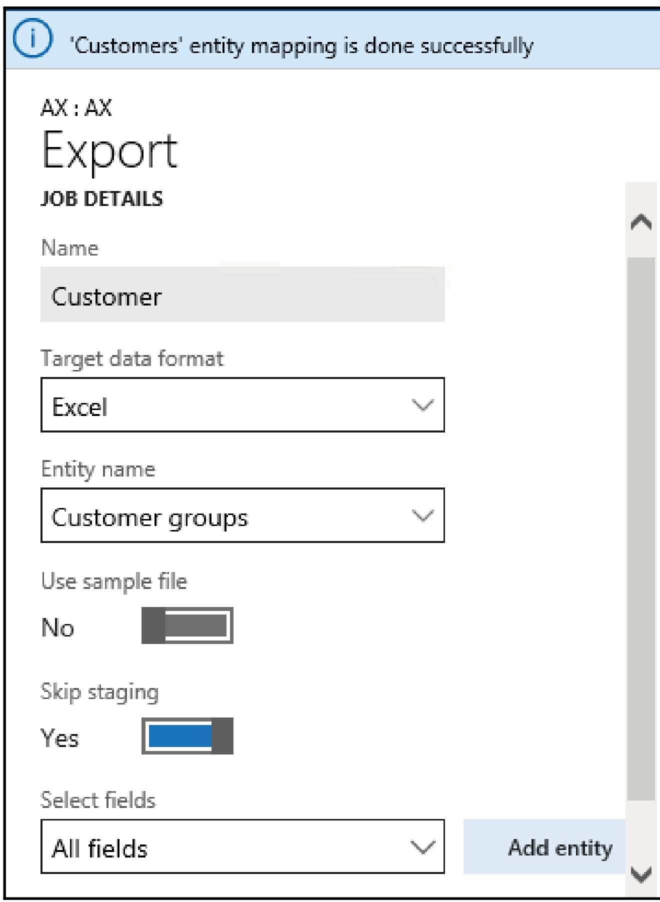
How to do it...

Let's create a simple package first that will contain two entities. Carry out the following steps in order to complete this recipe:

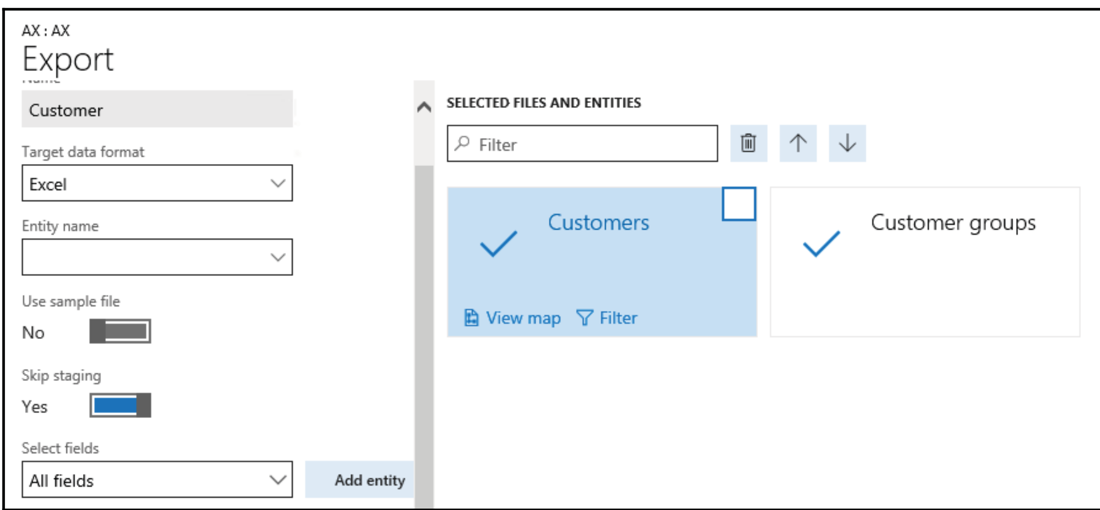
1. Navigate to **Workspace** | **Data management** | **Export.**
2. Fill out details as shown in the following screen, with your first entity **Customer**. Click on **Add Entity**:



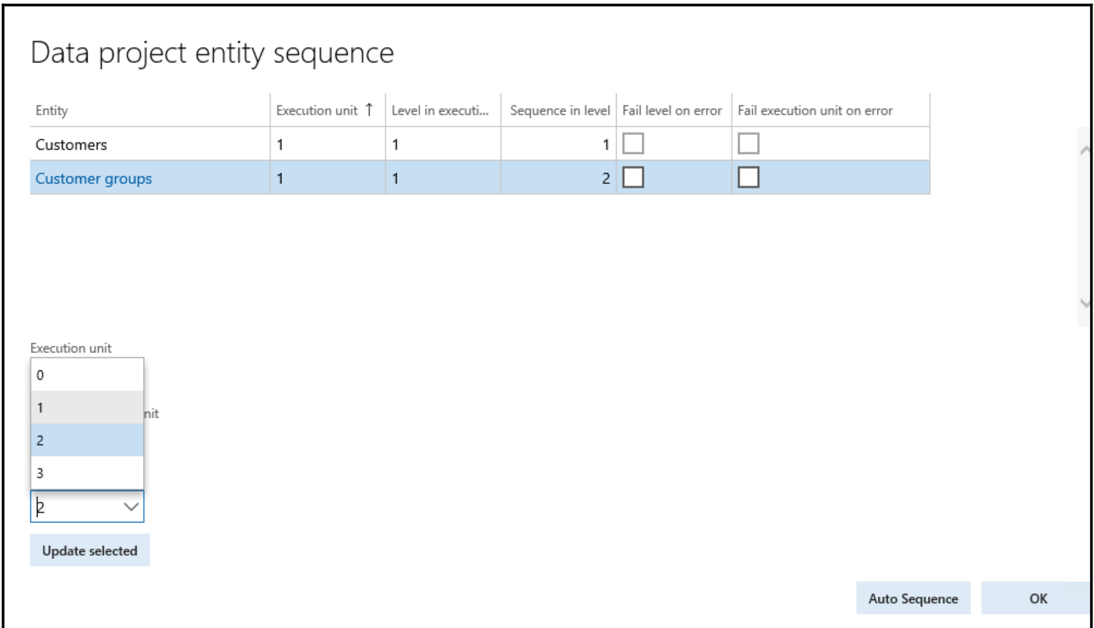
3. Add another entity in your package, **Customer groups**. Click on **Add entity**:



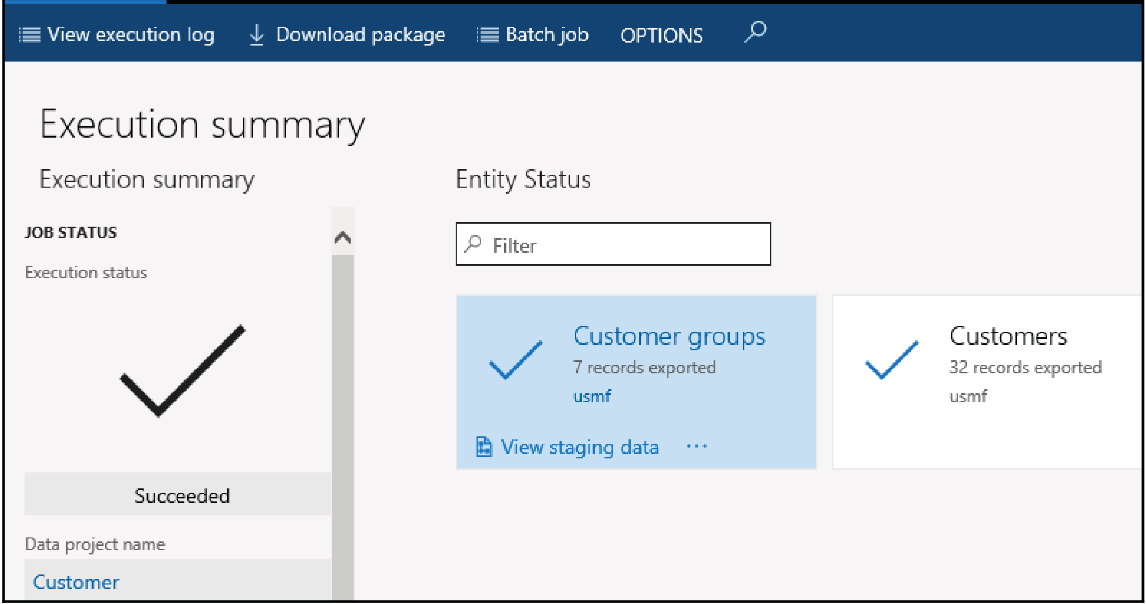
4. Your screen must look as follows:



5. Click on **Entity Sequence** and change the `Customer groups` sequence to 1, and click on **Update selected**. Now update the sequence for `Customers` to 2 and click on **Update Selected**. Now click on the **OK** button:



6. Now you are on the **Data Management** home screen. Click on the **Export** button. You will be redirected to the **Job Execution** form. Once execution has succeeded, click on **Download package**:

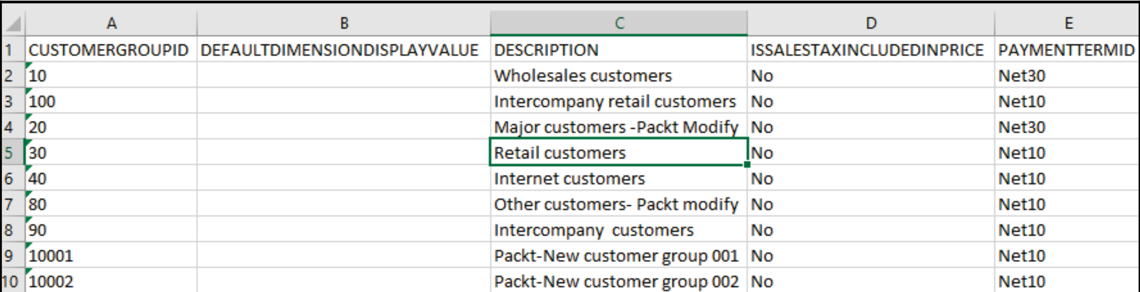


7. While asking for Data, the package does not exist. Click **Yes** to create a new package. You will get a ZIP file with both entities along with two more files, **Manifest** and **PackageHeader**. Both files have metadata for your package.
8. Now let us try to import this package with a few new records and some updates in existing records.

   In customer entity, modify entities as follows:

	A	B	C	D	E	F
1	CUSTOMERACCOUNT	NAME	ACCOUNTSTATEMENT	ADDRESSBOOKS	ADDRESSCITY	ADDRESSCOUNTRYREGIONID
2	US-028	Contoso Retail Miami-Packt	Always	RetailCust	Miami	USA
3	US-040	Contoso Retail USA-Packt	Always		Seattle	USA
4	US-041	Dolphin Wholesale-Packt	Always		Fargo	USA
5	US-100	Packt-New customer1	Always		Fargo	USA
6	US-101	Packt-New customer2	Always		Fargo	USA
7	US-102	Packt-New customer3	Always		Fargo	USA
8	US-103	Packt-New customer4	Always		Fargo	USA
9						

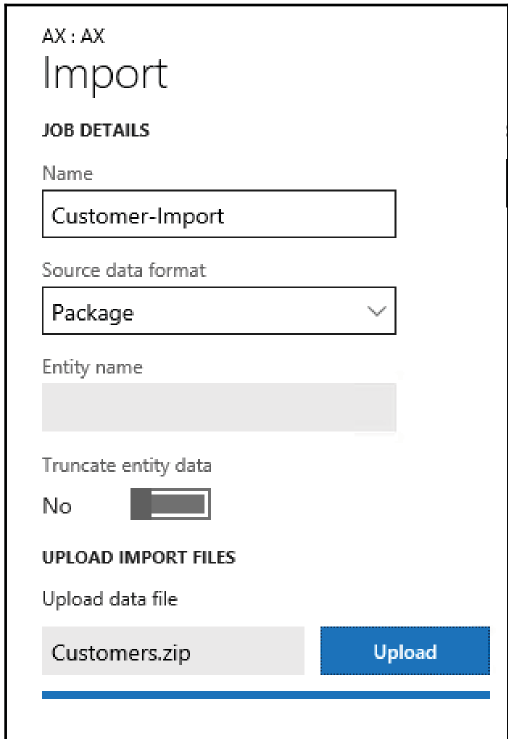
In customer group, modify entities as follows:



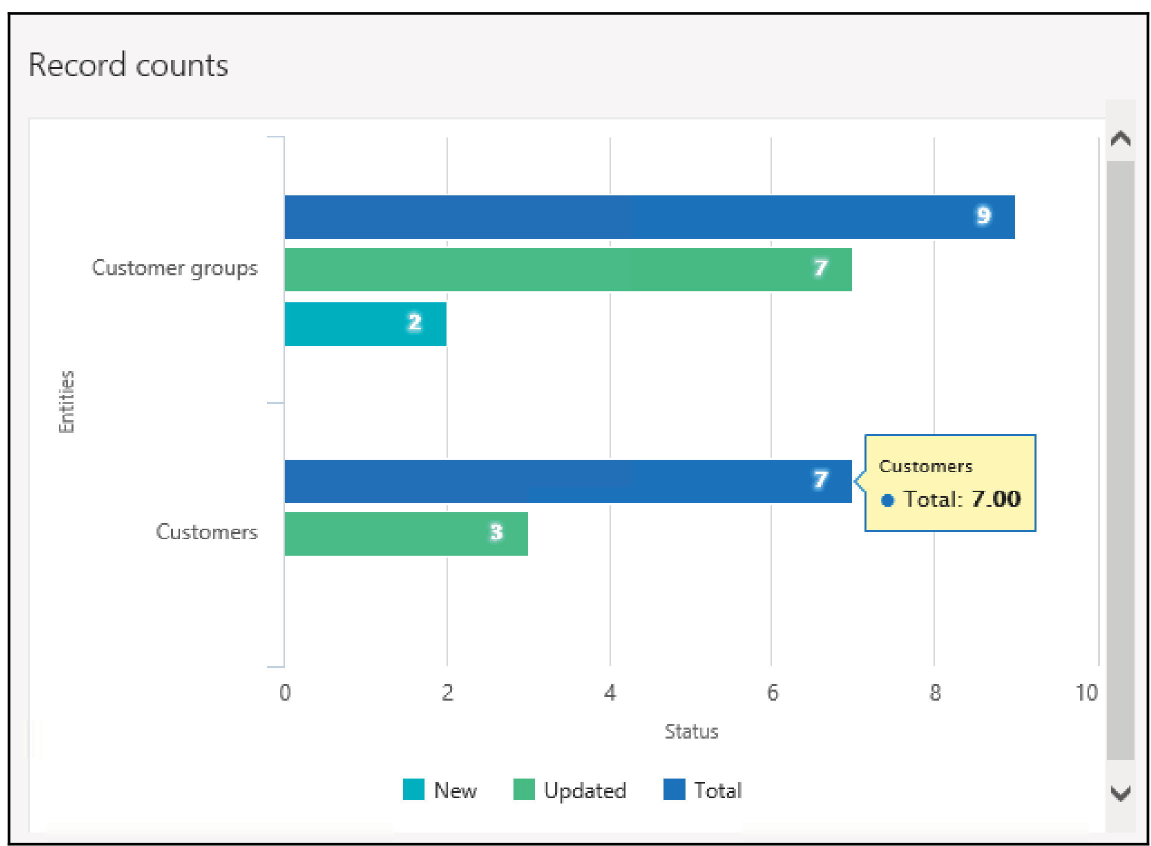
	A	B	C	D	E
1	CUSTOMERGROUPID	DEFAULTDIMENSIONDISPLAYVALUE	DESCRIPTION	ISSALESTAXINCLUDEDINPRICE	PAYMENTTERMID
2	10		Wholesales customers	No	Net30
3	100		Intercompany retail customers	No	Net10
4	20		Major customers -Packt Modify	No	Net30
5	30		Retail customers	No	Net10
6	40		Internet customers	No	Net10
7	80		Other customers- Packt modify	No	Net10
8	90		Intercompany customers	No	Net10
9	10001		Packt-New customer group 001	No	Net10
10	10002		Packt-New customer group 002	No	Net10

Save your data and make a ZIP file along with **Manifest** and **packageHeader**.

9. Now let's try to import this data package in the system. We have two options to import packages in a system. First using LCS tools and using data management workspace. In our recipe we will use data management workspace. Navigate to **Data management** workspace and select **Import**. Fill details in as follows:



10. Now click on the **Import** button. A job will execute and insert/update data in its respective table. You can see the record counts once it completes:

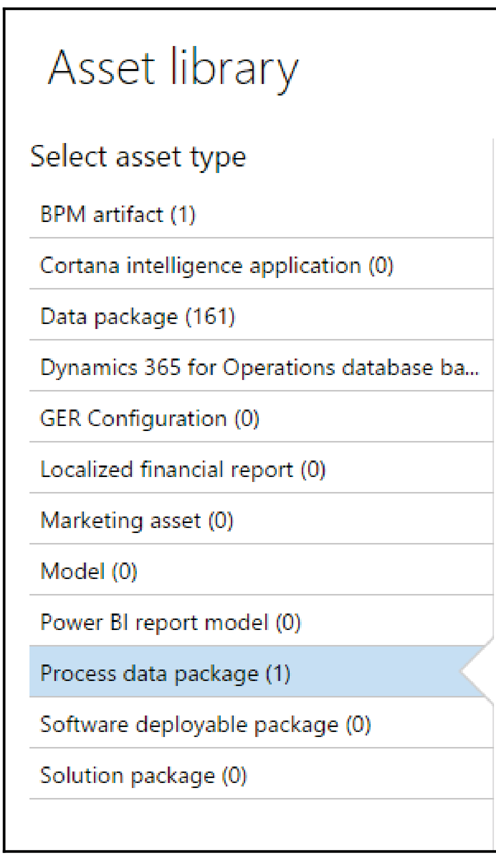


There's more...

With the LCS tool, you can create default packages as well using Process Data Package. LCS will create Data packages for you for all available modules with proper sequencing and mapping.

Let's see how to create default Data packages in LCS:

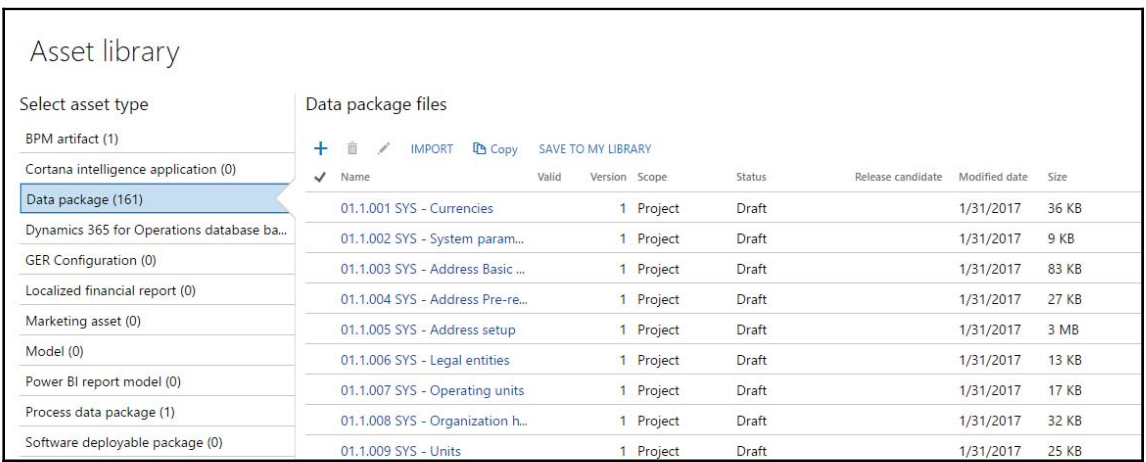
1. Log into your LCS account. Select your project in case you have multiple projects aligned with your LCS account.
2. Go to **Asset Library**, select **Process data package**:



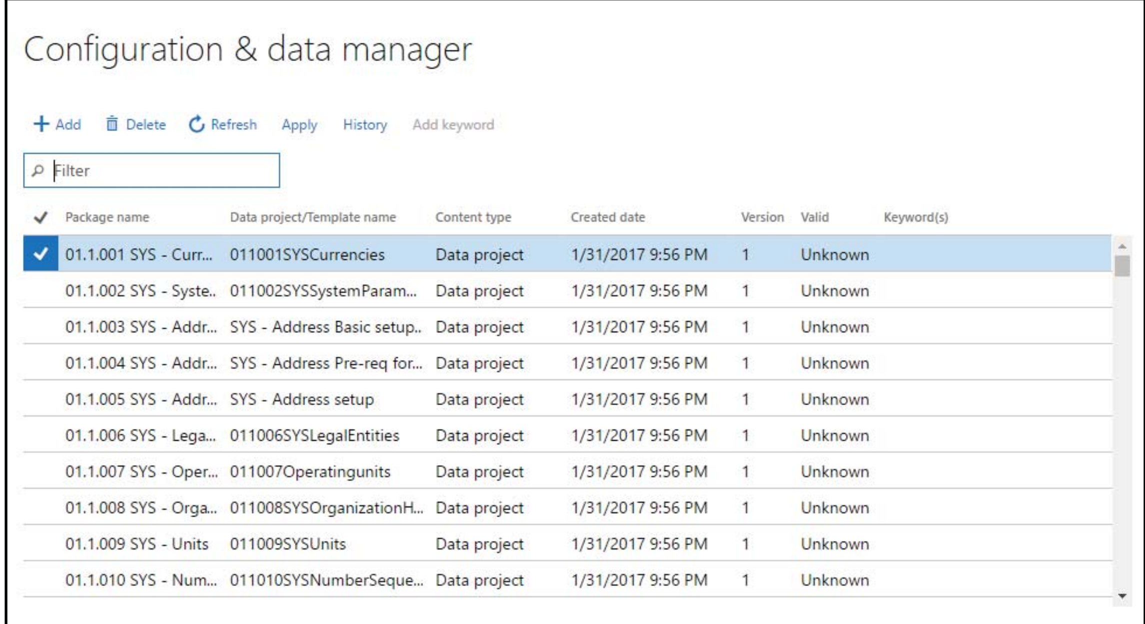
3. Click on the **Import** button and select a **Process Data Package** template and click on **pick**.

   It will create a new **Process data package** file with the status, Processing. It will take a few minutes to create all packages under the **Data package** tab.

4. Once done, refresh your page and select **Data package** under **Asset Library**, you will find that all default packages are added here:



5. The same will be added in the **Configuration & data manager** tool. To check, go back to the Project home screen in LCS and select the **Configuration and Data Manager** tool. Your screen must look as follows:



6. From here you can download the packages and upload them with real data. Once you have all required data in entities, the same package can be uploaded in Dynamics 365 for Finance and Operations through the **Configuration & data manager** tool.

See also

- You can find all available Data packages with their sequence number at the given link: `https://docs.microsoft.com/en-us/dynamics365/unified-operations/dev-itpro/data-entities/data-entities-data-packages`.

Data migration

Data migration is a key task of a project implementation cycle. Here are the key points regarding export/import tasks.

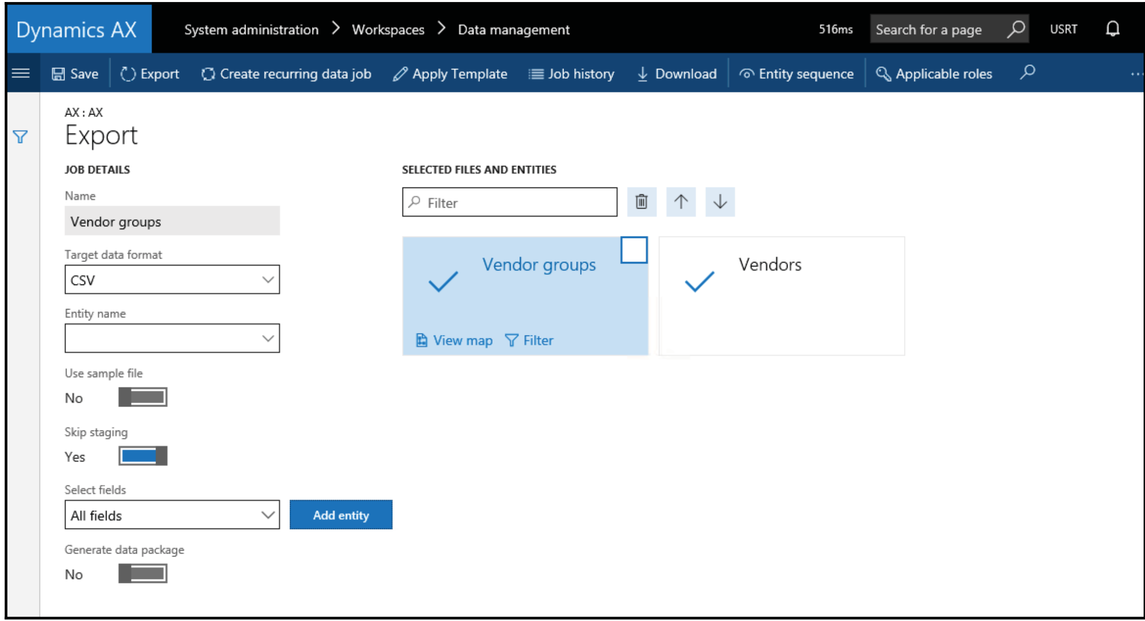
The following pain points can occur during migration:

- Inability to do quick iterative migration and validations
- Multiple hops that lead to multiple dependencies and change of errors
- Complexity due to repeated manual interventions
- Difficulty in tracing and error troubleshooting
- Difficulty migrating a large volume of data within a time constraint

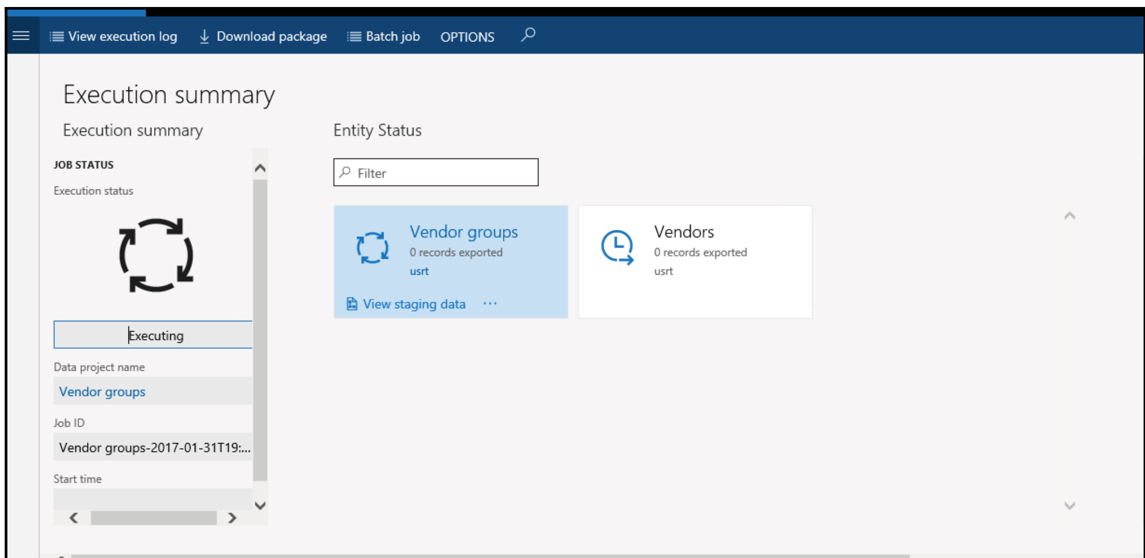
During migration, you can strategize and choose data entities. Data entities also save time during implementation because previous activities required data export from a database, data export validation, and data transformation to files such as Excel or XML. In the current version of Dynamics 365 for Finance and Operations, these hops have been eliminated. If an import error occurs, you can skip selected records and choose to proceed with the import using the good data, opting to then fix and import the bad data later. You will be allowed to partially continue and bad data will be indicated by using errors. Data imports can be easily scheduled using a batch, which offers flexibility when it is required to run. For example, you can migrate customer groups, customers, vendors, and other data entities in the system at any time.

Getting ready

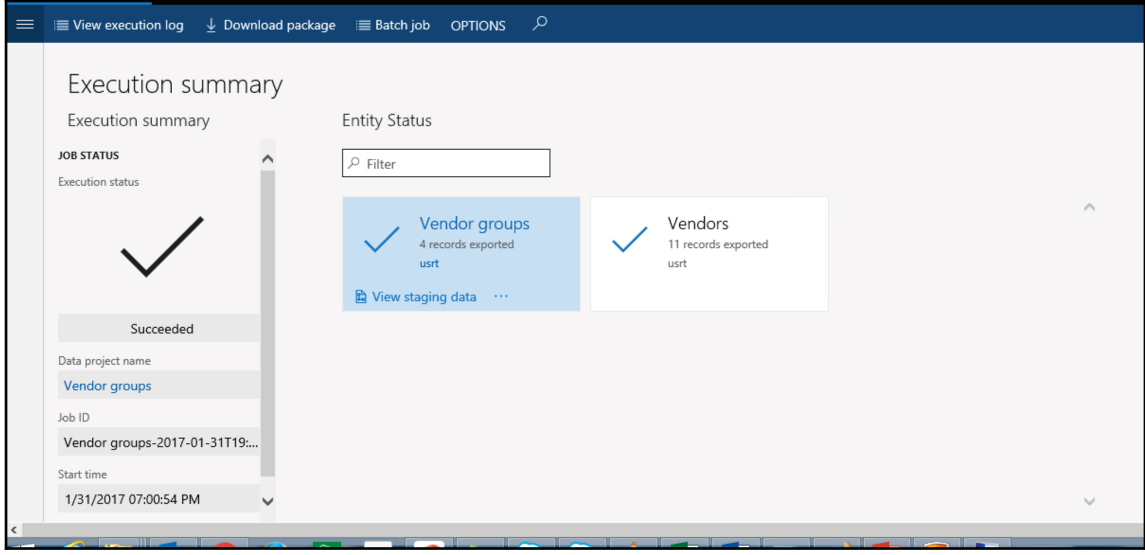
Export is the process of extracting data from the system utilizing data entities. The export process is done through a project. When creating the process, there is a lot of flexibility as to how the export project is defined. There is an option to choose not only which data entities to export, but also the number of entities, the file format used (there are 14 different formats to choose for export), and apply a filter to each entity to limit what is exported. After the data entities have been pulled into the project, the sequencing and mapping described earlier can be performed for each export project.



After the project is created and saved you can export the project to create a job. During the export process, you can see a graphical view of the status of the job and the record count. This view shows multiple records so you can review the status of each record prior to downloading the actual files.



After a few seconds, click **refresh** and you will notice a change in status of the job:

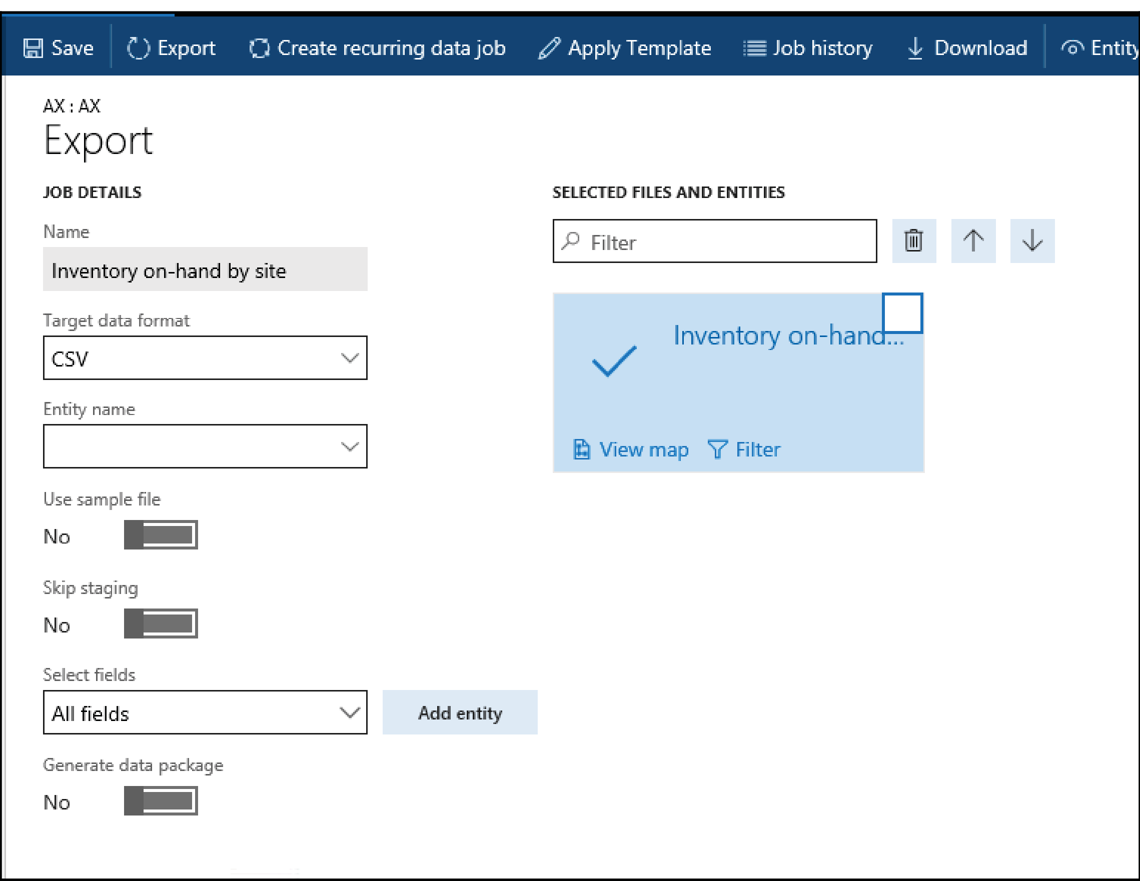


After the job is completed, you can choose how to download the files of each data entity either as a separate file or by combining the files as a package. If there are multiple data entities in the job, choosing the package option will expedite the upload process. The package is a zipped file, containing a data file for each entity as well as a package header and manifest. These additional documents are used when importing in order to add the data files to the correct data entities and sequence the import process.

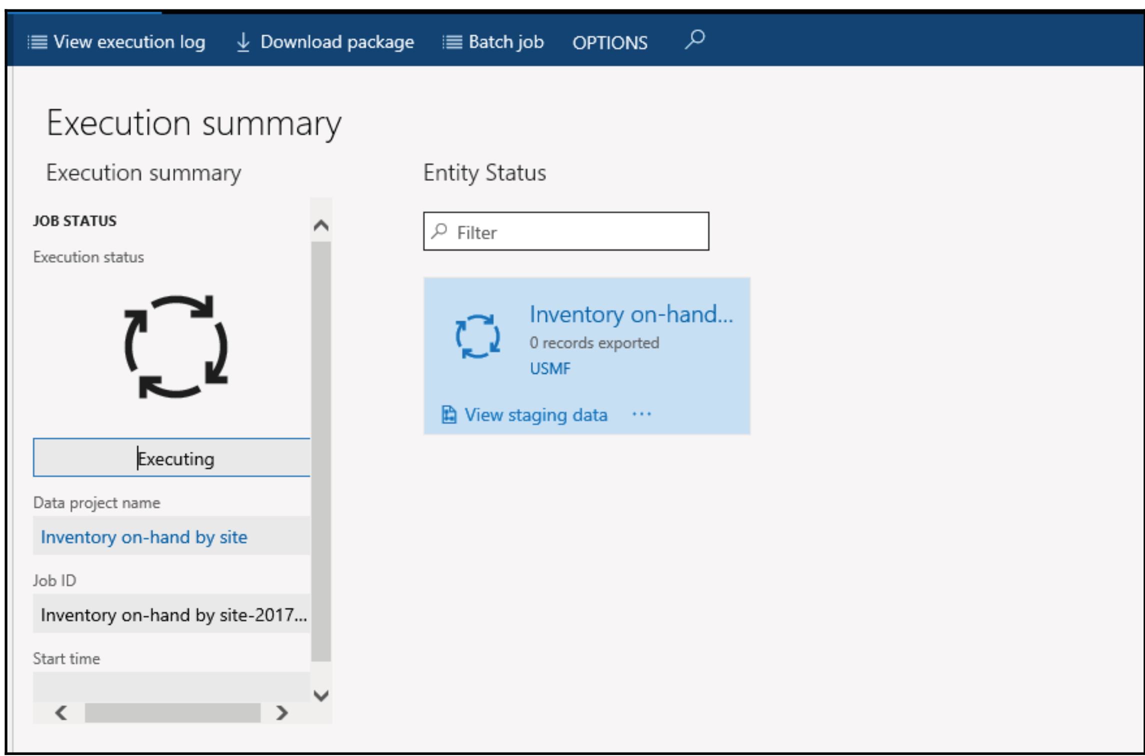
How to do it...

Dynamics 365 for Finance and Operations provides out-of-the-box entities for data export and import; we will take an example to export stock on hand by inventory site:

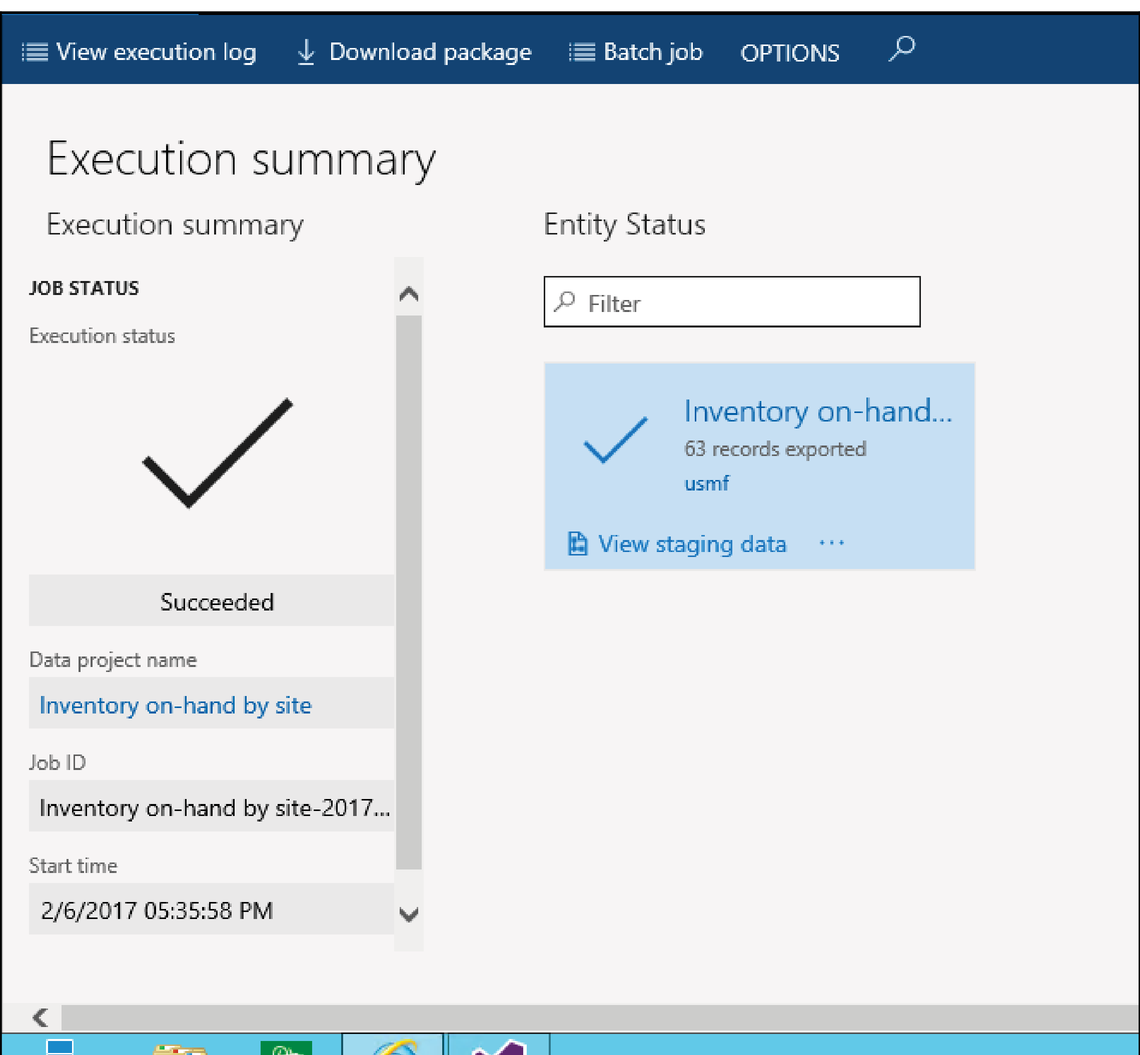
1. Click on **Export** in data management:



2. Click on the **Export** button to export the data and to schedule a batch job select **Create recurring data job**. After clicking **Export** it will show the following screen:



3. On the refreshing form, we see status as **7426** records exported. Here we need to click **Download package** to download the data here and the system will create a data package for us:



You can check the staging logs for details by navigating to the **View staging data** button, as shown in the preceding screenshot.

How it works...

In normalized tables, a lot of the data for each item might be stored in an Invent Sum table, and then the rest might be spread across a small set of related tables. Dynamics 365 uses the data entity for the inventory on hand concept which appears as one de-normalized view, in which each row contains all the data from the invent sum table and its related tables to show inventory stock by site. An export job gives feasibility to either export the data from target entity to staging table and then write the same data to the file or skip staging and write the data directly to file. Here, when we choose staging then the system writes the data to file using SQL server integration services.

Importing data

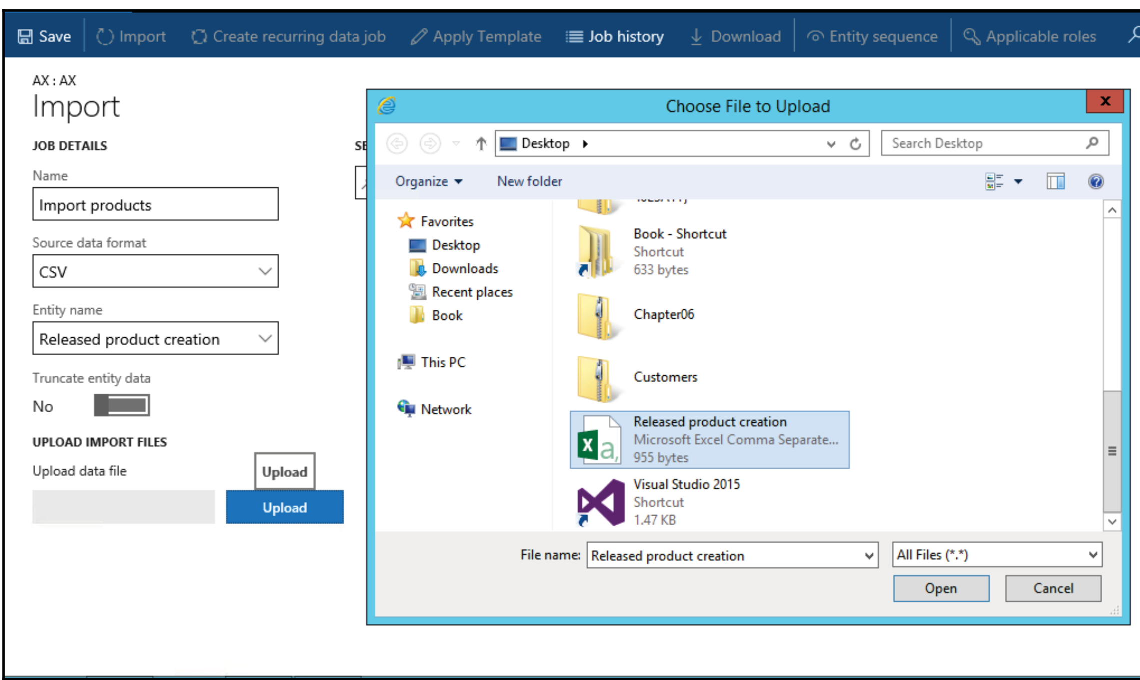
Import is the process of pulling data into a system utilizing data entities. The import process is done through the **Import tile** in the **Data Management** workspace. Data can be imported either for individual entities or for a group of logically related entities that are sequenced in the correct order. The file formats vary depending on the type of import. For an entity, it can be an Excel file that is comma-separated, tab-separated, text. For a data package, it is a ZIP file. In both cases, the files are exported using the previously-mentioned export process. The detailed steps for importing data using data packages are as follows.

How to do it...

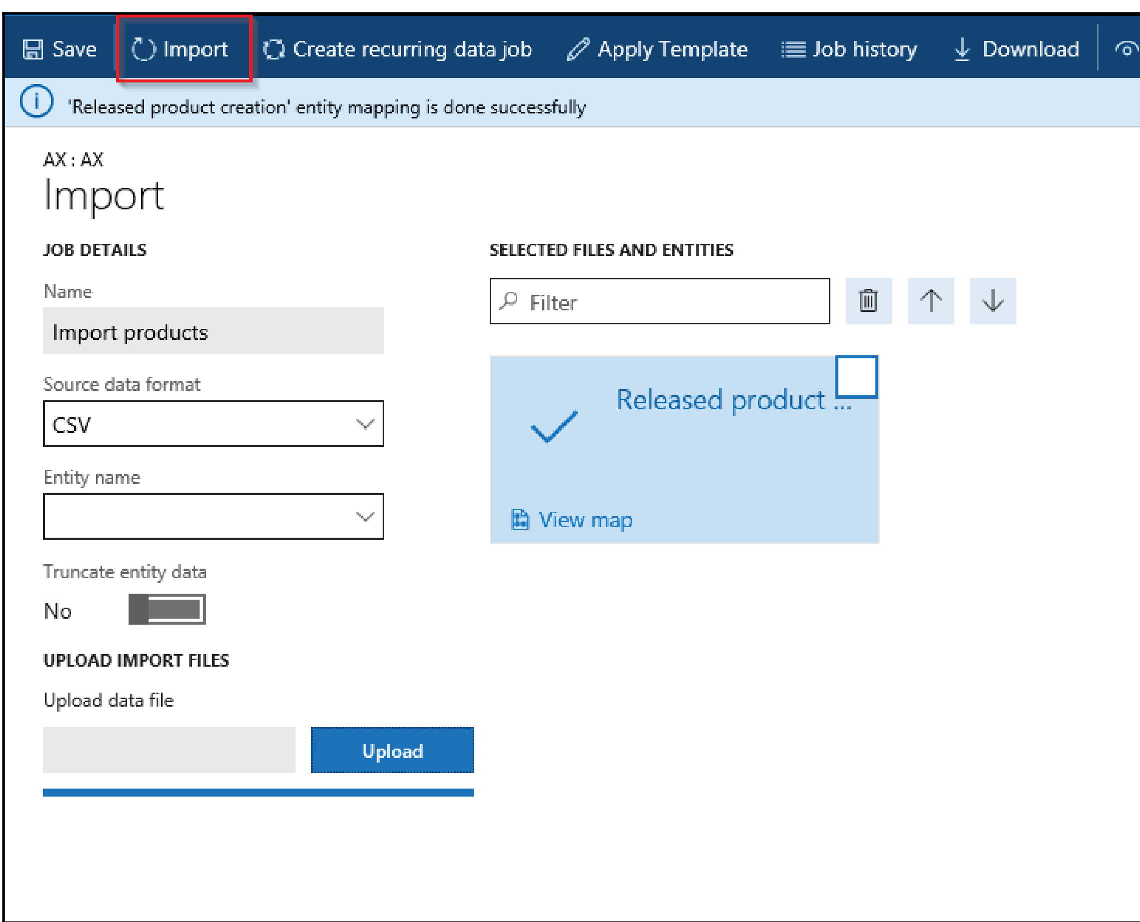
Let us import and create released products in Dynamics 365 for Finance and Operations:

1. In the system administration module, click **Data management workspace**, to begin importing, select **import tile**.
2. In the **Name** field, provide a logical name for the package, which is being imported. In the **Source Data Format** field, select `CSV` as the source data format.

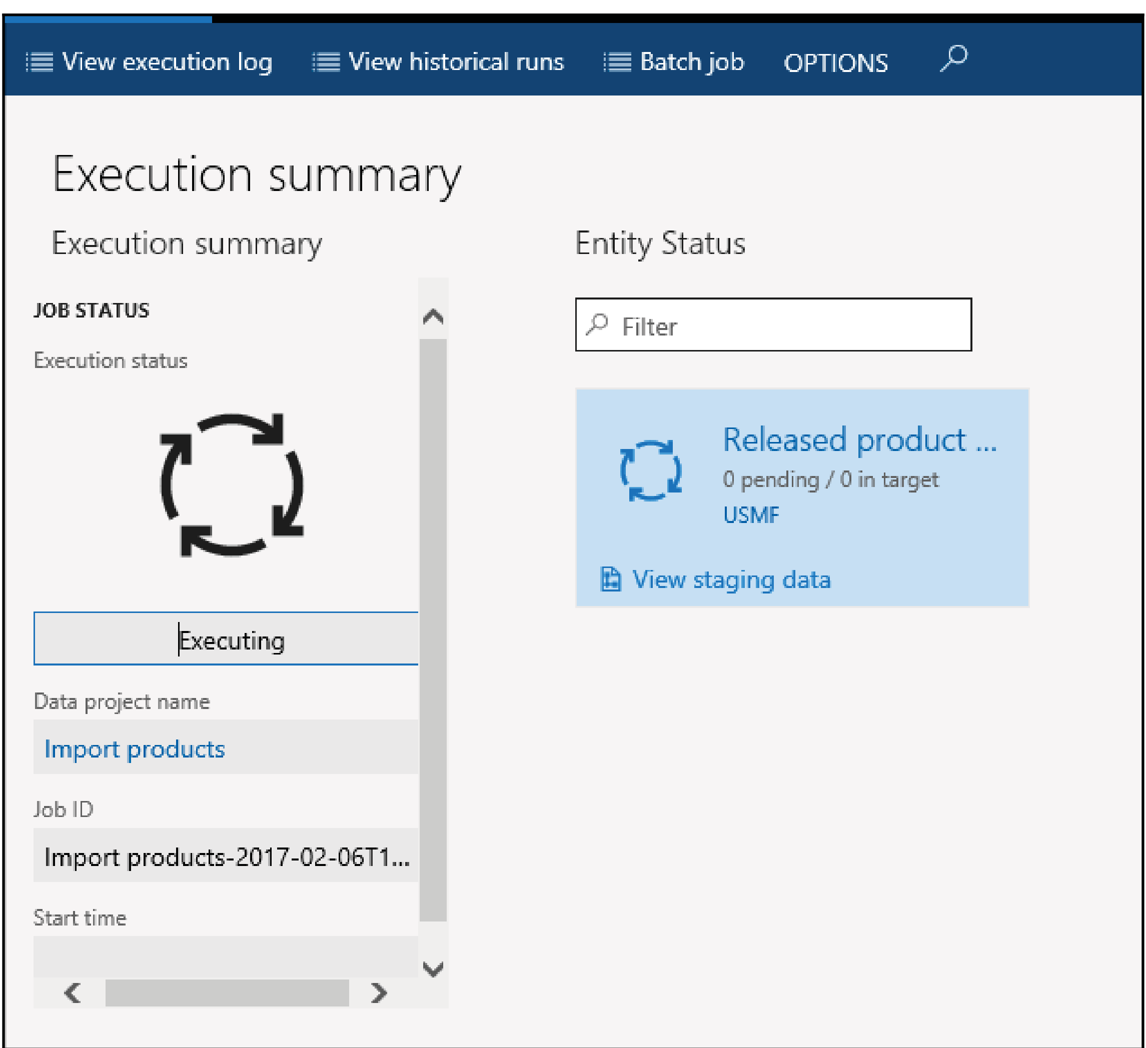
3. Click the **Upload** button and choose the file from the location for the data that is being imported.



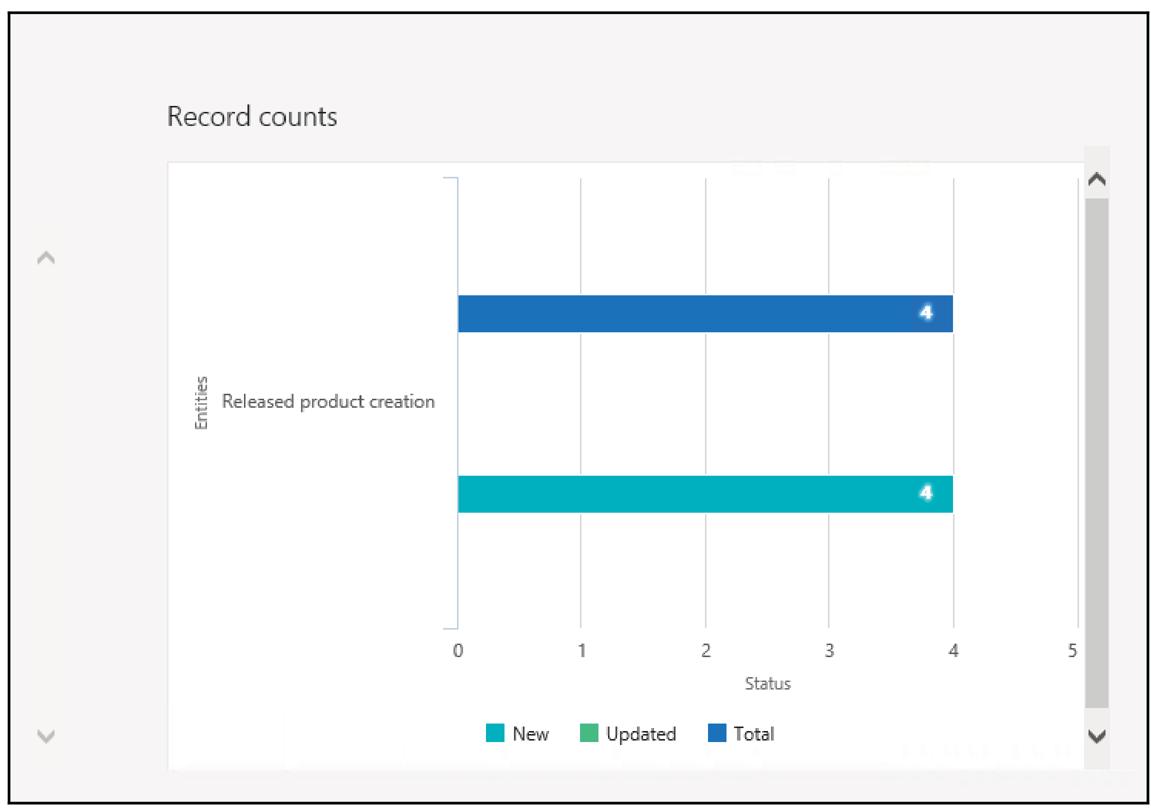
4. After clicking **Import**, the system will import the file in a blob, as shown in the screenshot:



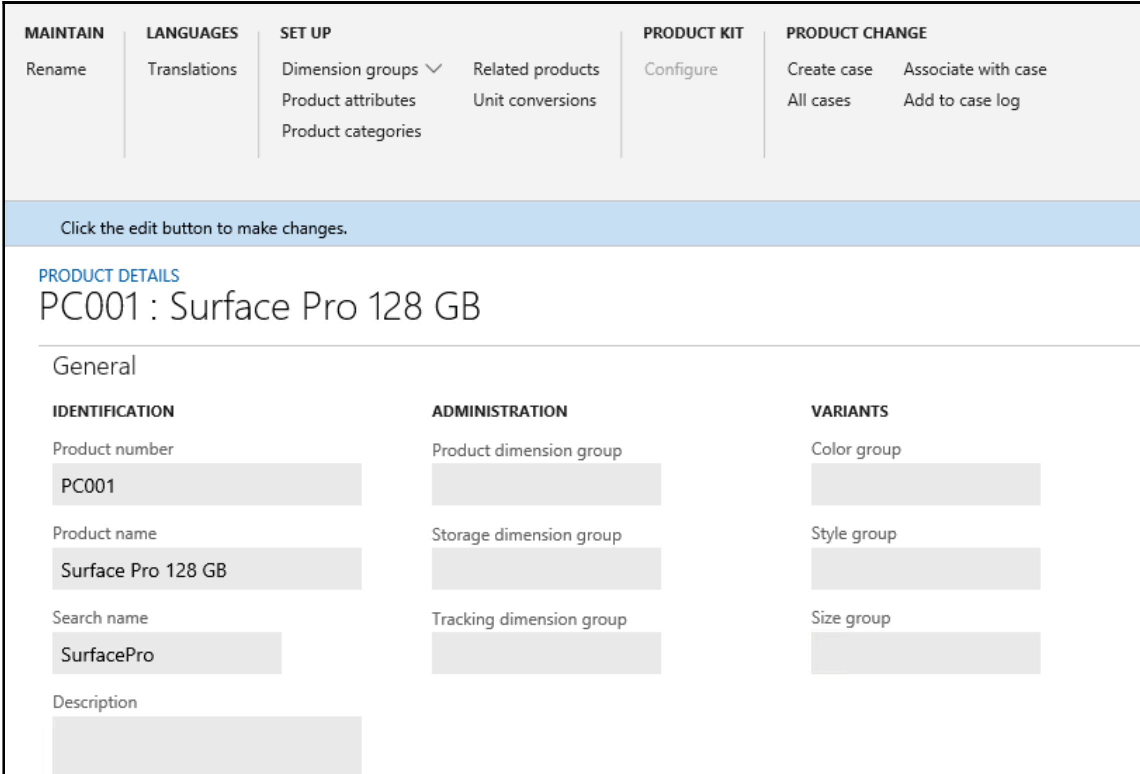
5. Click **Refresh** to update the job status:



6. Check the status of the job to see if the records are imported:



7. View staging data and check the product in released form:



Finally, we could check that the product imported by us is released in designated company. Using data management platform in Dynamics 365 for Operation we could import thousands of products and release them in multiple companies.

How it works...

Dynamics 365 creates a staging table for each entity in the database where the target table resides. Data that is being migrated is first moved to the staging table. There, you can verify the data, and perform any cleanup or conversion that is required. You can then move the data to the target table or export it. Here in, we provide a source file for import of released products, then the system imports the flat data file in a staging table using the SSIS package and then uses X++ jobs to move data from the staging table to the target table to create records in `EcoResProduct`, `InventTable`, and related tables. It uses data entity for insertion of flat data from staging in target tables so that related records are created and their integrity is maintained.

Troubleshooting

Now you must have understood the concept and logic behind data management. We also created a few entities, packages, and other related objects. Import and Export become very easy with Dynamics 365; however, these processes are not always straightforward. You may get a few unwanted errors during this.

In this section, we will see how to troubleshoot these errors and warnings.

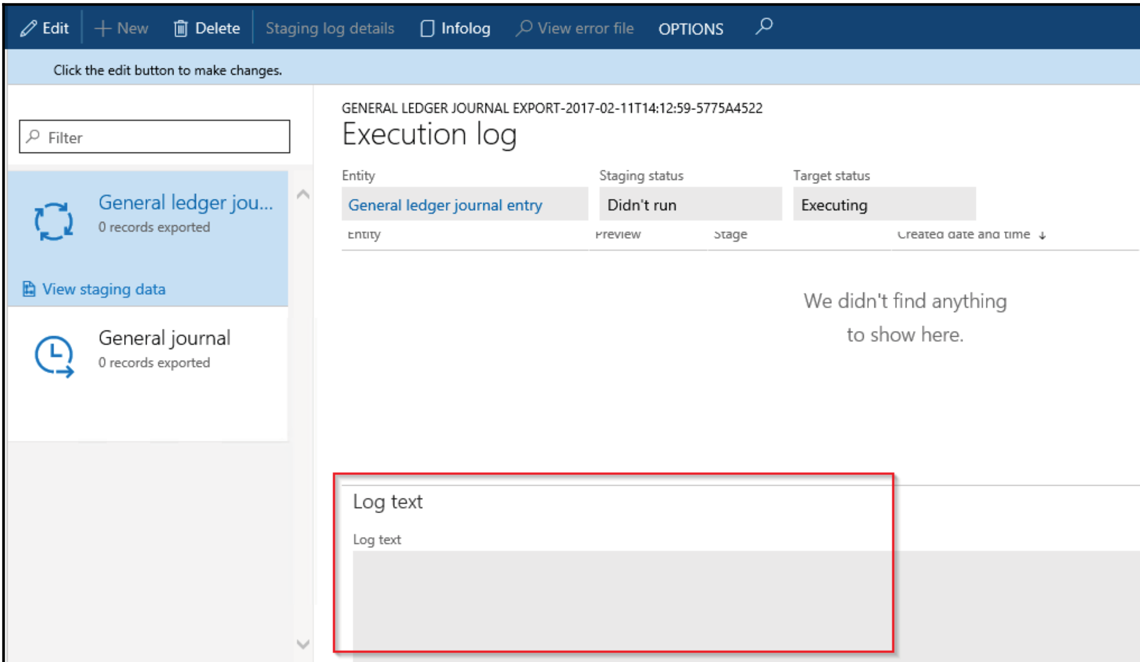
Getting ready

While working with data management at times we encounter errors in import and exports. The framework provided with the product is robust enough to identify the error logs and suggest for possible solutions to rectify the data. This section describes how to troubleshoot during the different stages of data package processing.

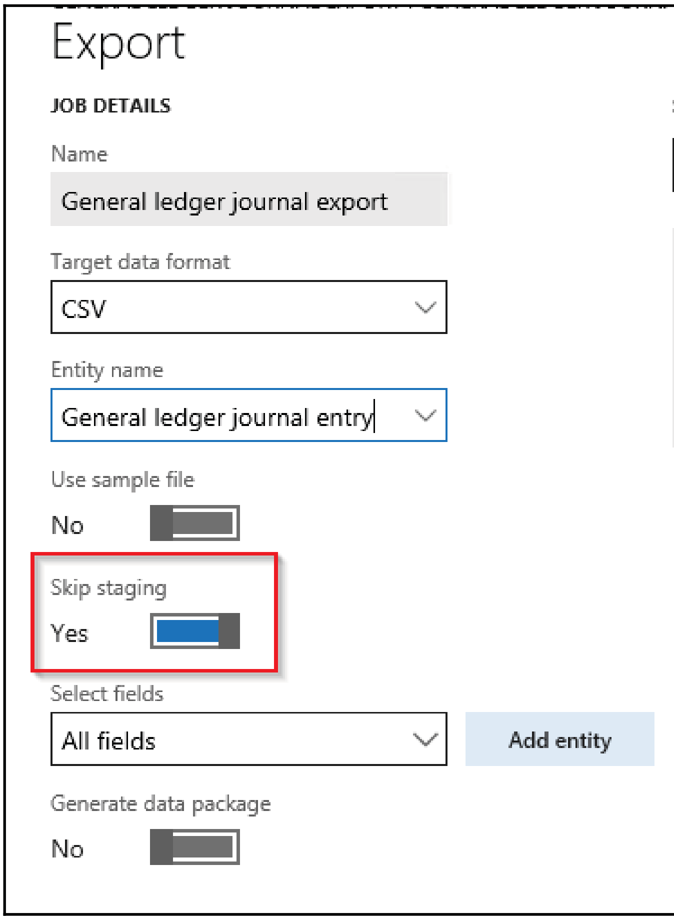
How to do it...

Carry on the following steps to troubleshoot some issues that you may get during routine development:

1. Normally during export processes, we do not get any errors, but if you get an error during the export process click **View execution log** and review the log text, staging log details, and **Infolog** for more information:

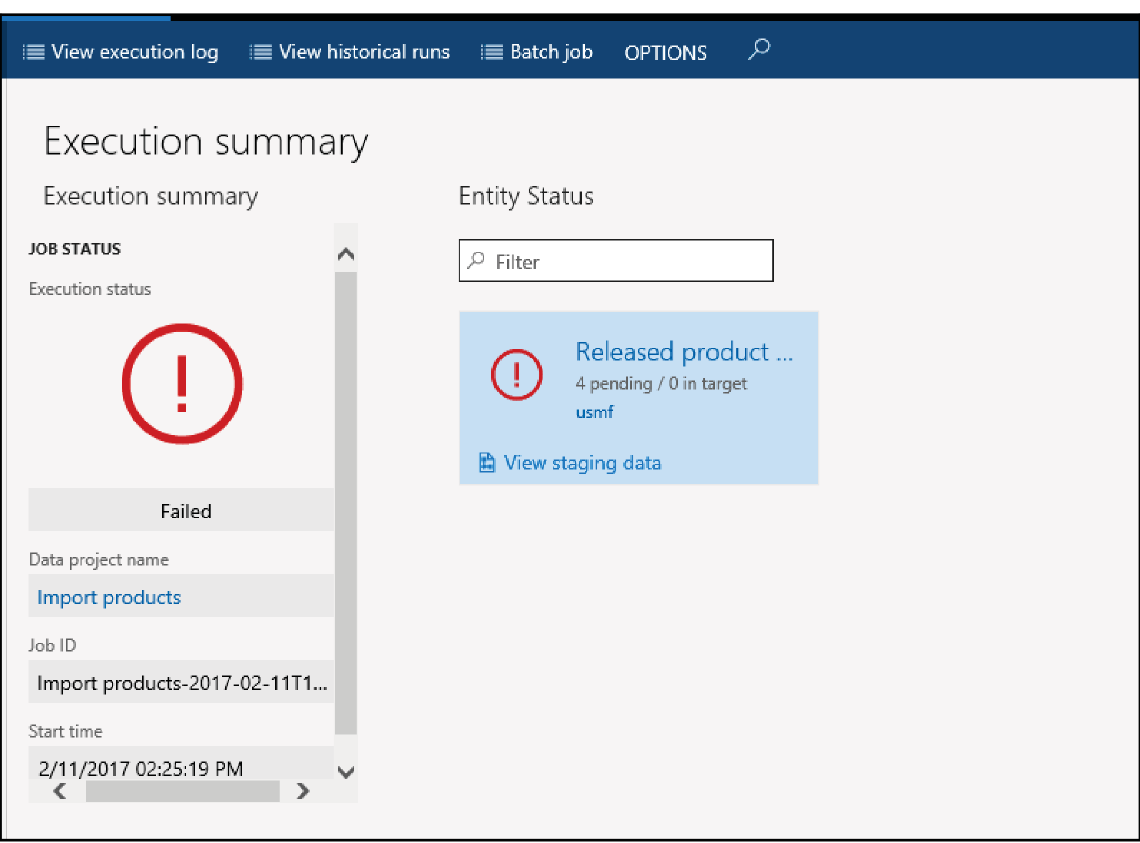


2. If you get an error during the export process with a note directing you to not skip staging, turn off the **Skip staging**, and then add the entity. If you are exporting multiple data entities, you can use the **Skip staging** button for each data entity:

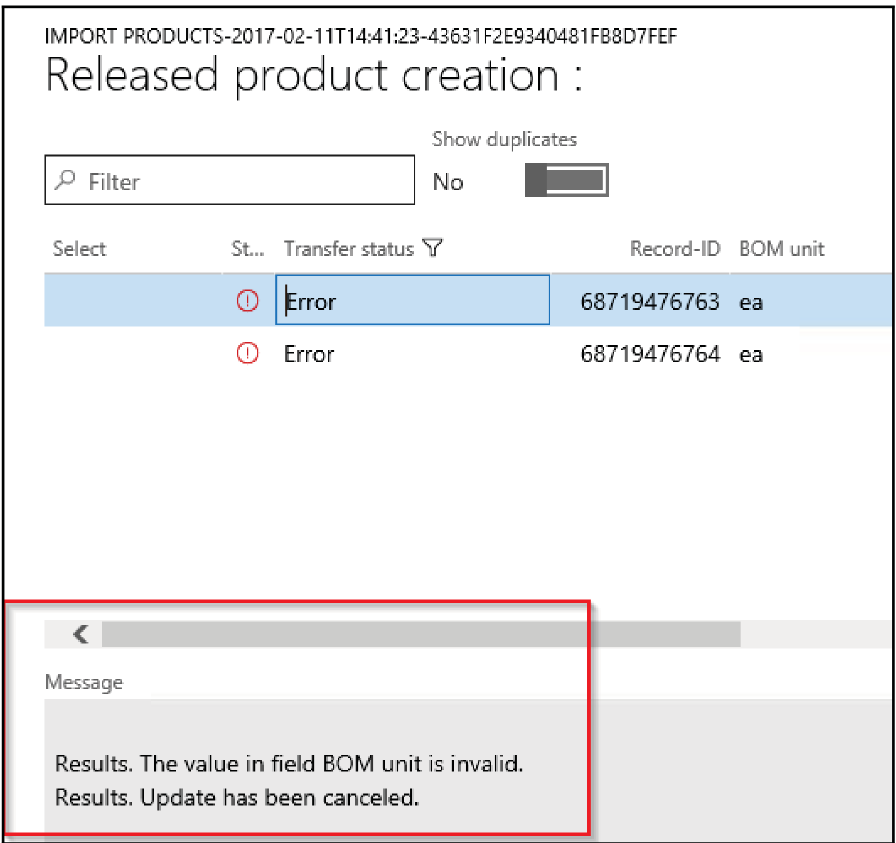


3. For carrying out import processes upload the data entity files.
4. If data entities do not display in **Selected Files and Entities** after you click **Upload** during the import process, wait a few minutes, and then check if the `OLEDB` driver is still installed. If not, then reinstall the `OLEDB` driver. The driver is `Microsoft Access Database Engine 2010 Redistributable – AccessDatabaseEngine_x64.exe`.

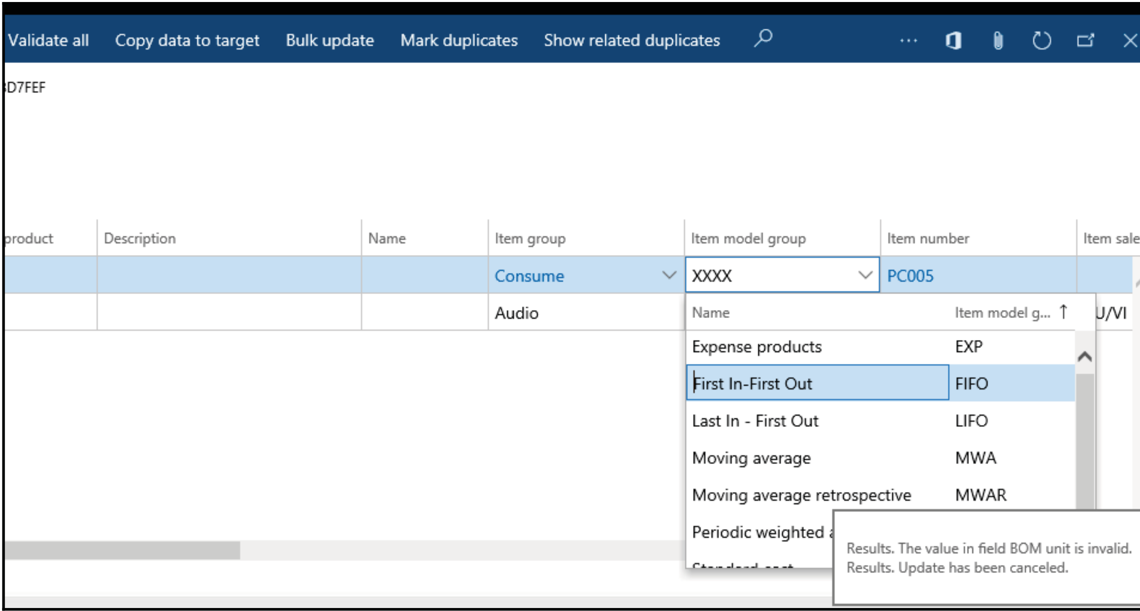
5. If data entities display in **Selected Files and Entities** with a warning after you click **Upload** during the import process, verify and fix the mapping of individual data entities by clicking **View map**. Update the mapping and click **save** for each data entity.
6. Import the data entities:



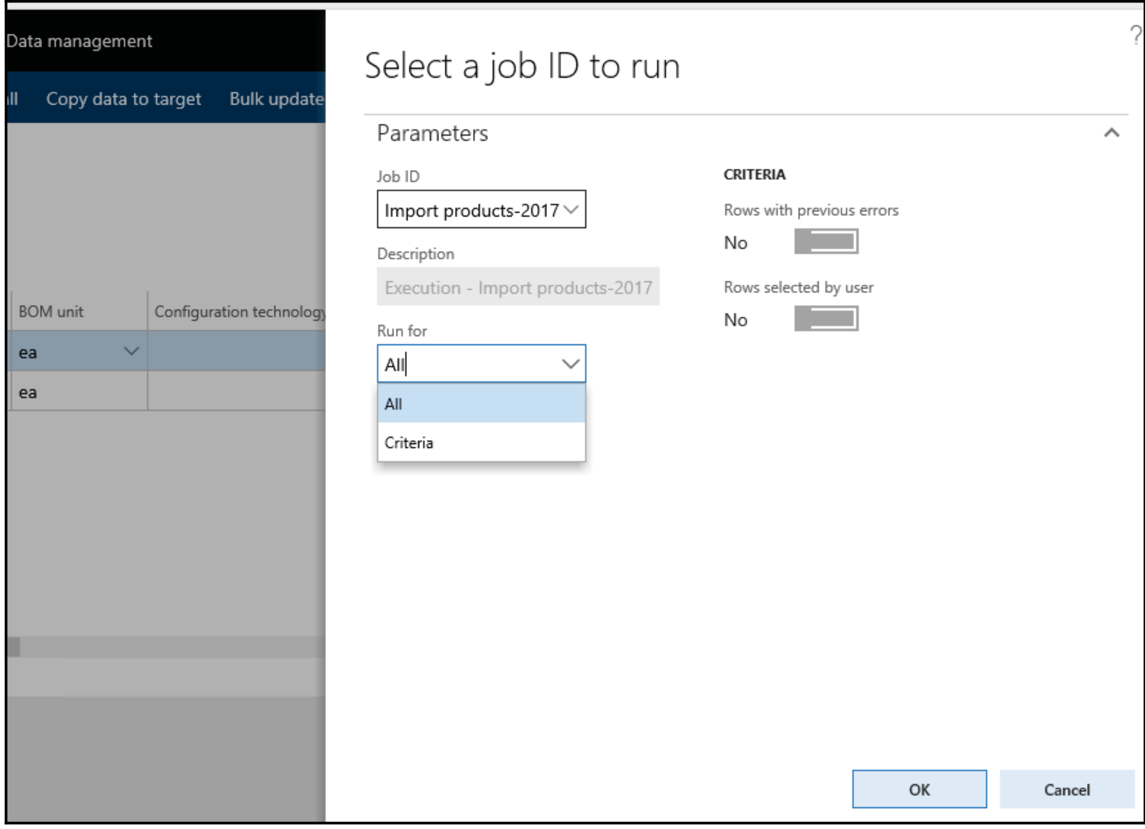
7. If data entities fail, (shown with a red X or yellow triangle icon on the data entity tile) after you click **Import**, click **View staging data** on each tile under the **Execution summary** page to review the errors. Sort and scroll through the records with **Transfer status** is equal to **Error** to display the errors in the **Message** section. Download the staging table via Microsoft Office. Fix a record (or all records) directly in staging by clicking **Edit, Validate all, and Copy data to target**, or fix the import file (not staging file) and re-import the data:



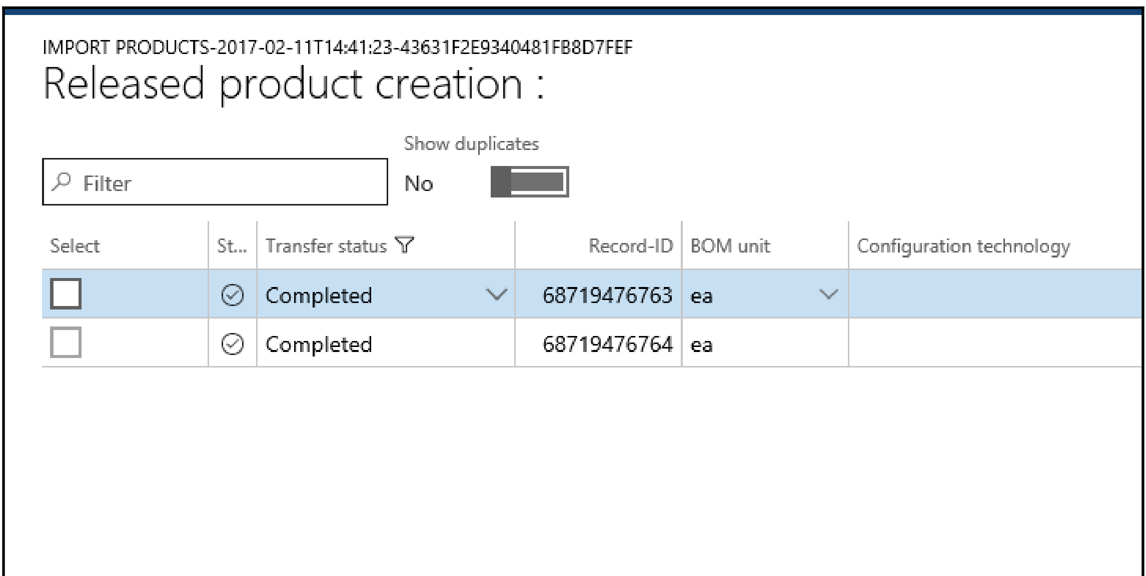
8. We could choose correct values in staging from the dropdown and then validate the lines:



9. After successful validation we could copy the data to target and run the job synchronously or asynchronously.



10. After the job is completed, we could check status of staging records as follows:

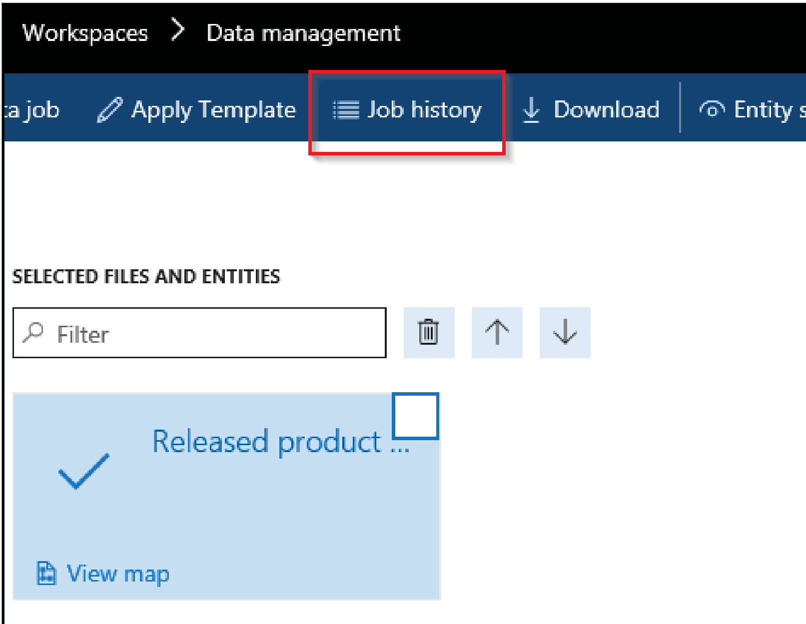


11. If data entities fail, you can check the import file to see whether there is an extra line in the file with text that displays "This is a string that is inserted into Excel as a dummy cell to make the column to support more than 255 characters. By default, an Excel destination component will not support more than 255 characters. The default type of Excel will be set based on the first few rows". This line is added during data export. If this line exists, delete this line, re-package the data entity, and try to import.

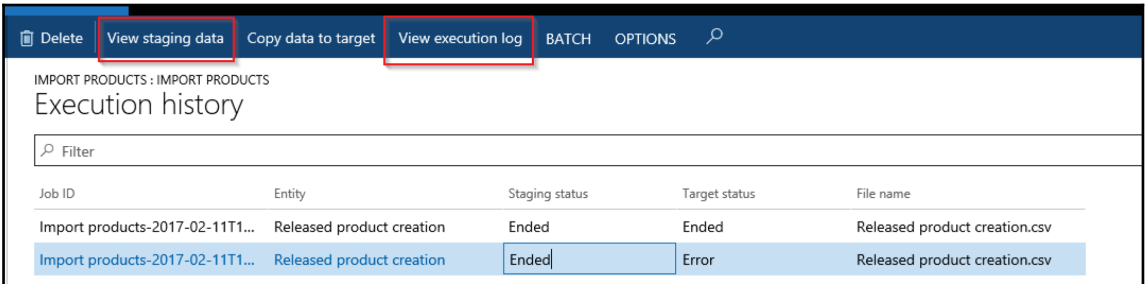
How it works...

Usually the system provides enough support to check the logs of error and resolve them. Let's see how we could check the log and interpret it to resolve the issue:

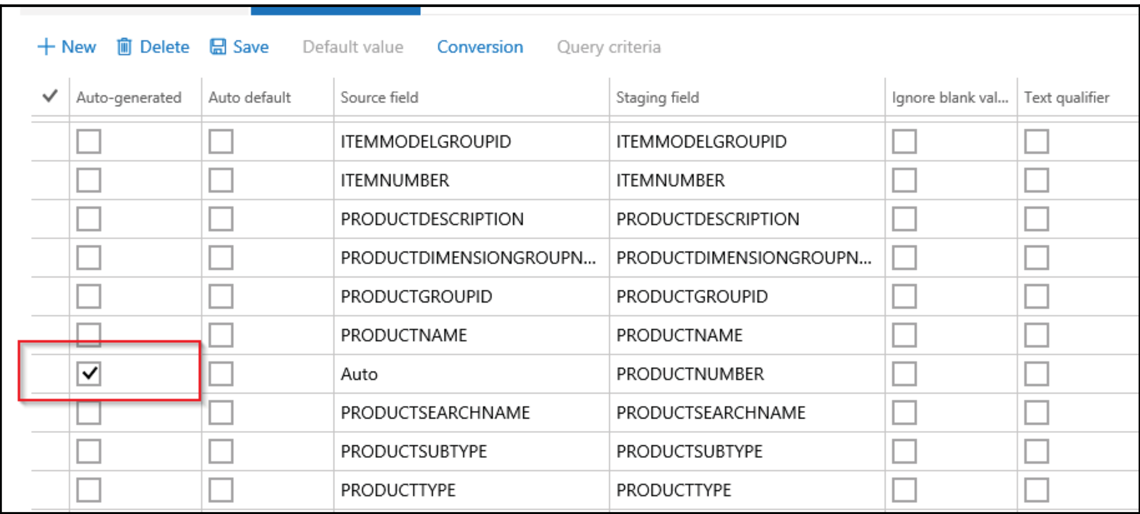
1. Status and error details of a scheduled job can be found under the **Job history** section in the **Data management** form:



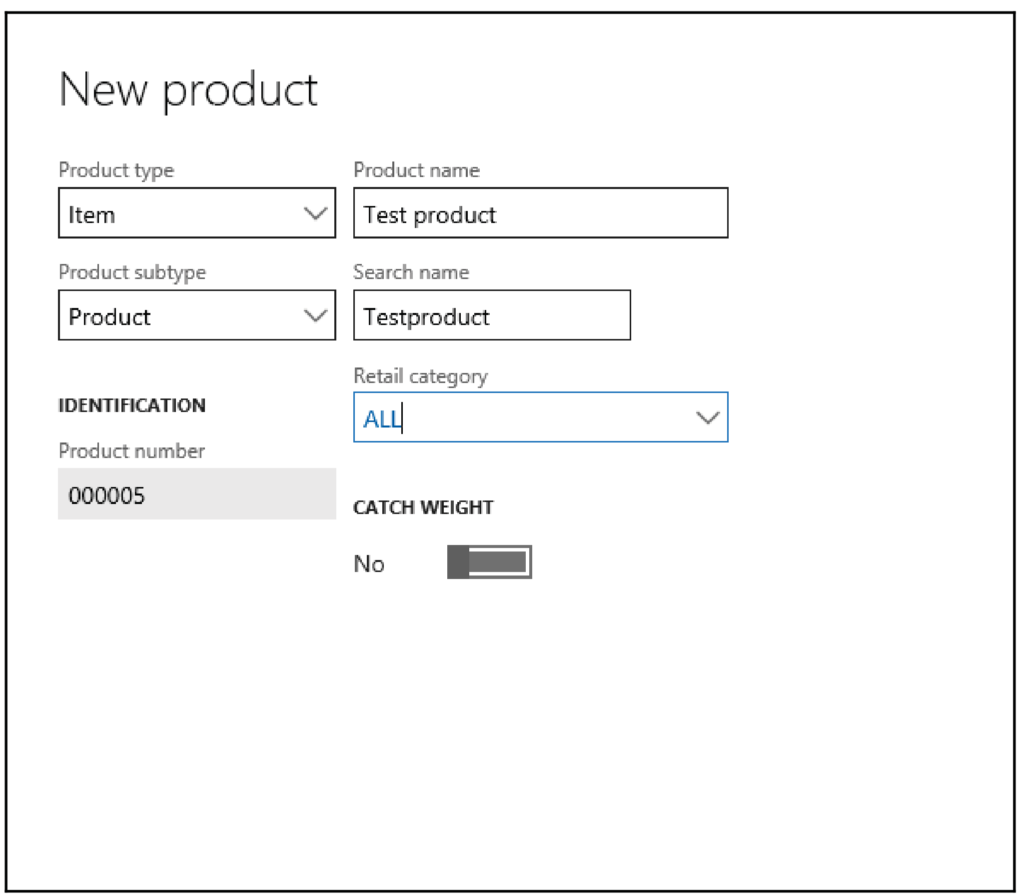
2. Status and error details of previous runs for data entities can be displayed by selecting a data project and clicking **Job history**. In the **Execution history** form, select a **job**, and click **View staging data** and **View execution log**. The previous runs include data project runs that were executed as batch jobs or manually:



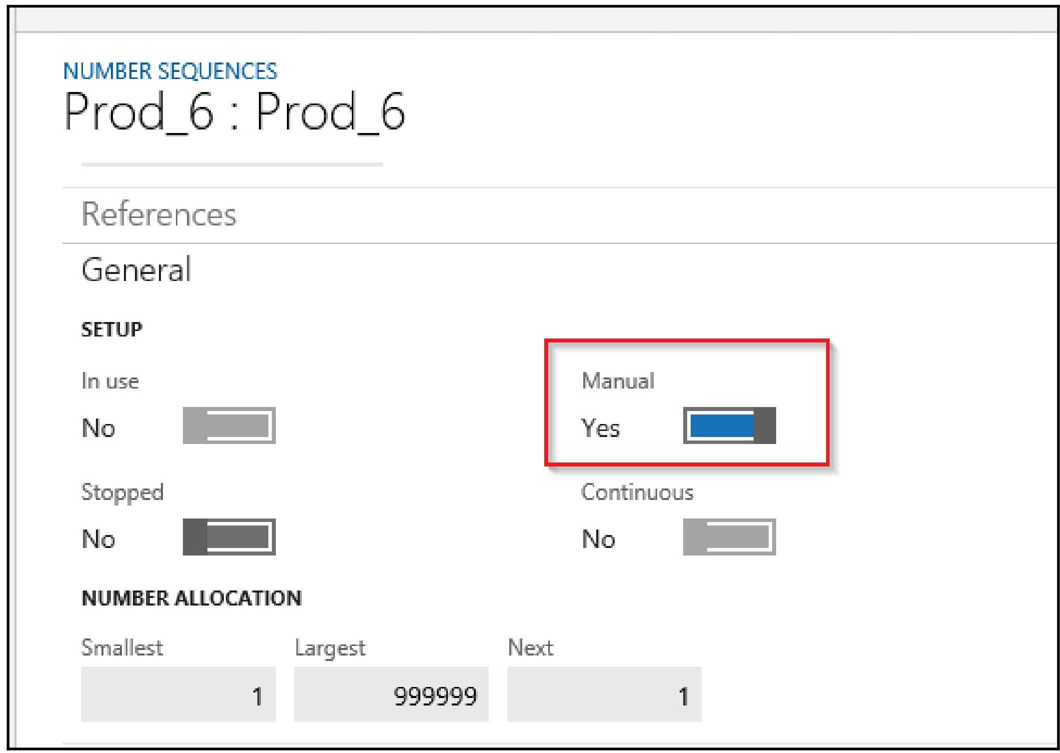
3. Many entities support automatic generation of identifiers based on number sequence setup. For example, when creating a product, the product number is automatically generated and the form does not allow you to edit values manually:



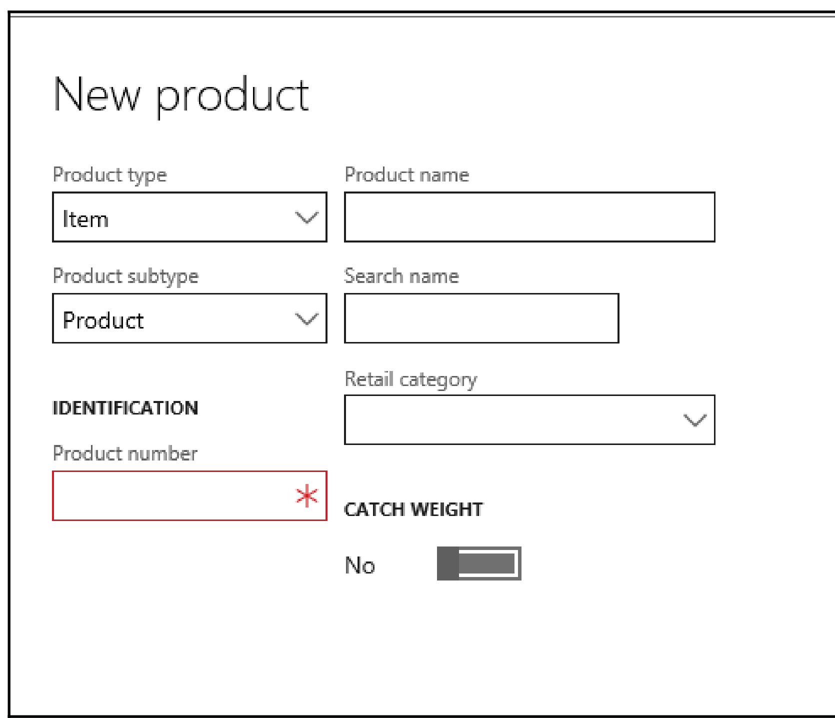
4. Here auto-generated fields will allow the framework to use the number sequence that auto generates the **Product number** and we need not include the product number in our import file:



5. It is possible to enable manual assignment of number sequences:



6. In this case, you can manually provide the value and enable manual assignment of product numbers instead:



7. If you are using the import file to create released products then you need to supply a product number column.

There's more...

While importing the system users entity, you may receive a integrity violation error if there is a guest user in the exported package. The guest user must be deleted from the package in order for the entity to work.

If a record already exists in the **UserInfo** table (the admin record would most likely always exist), the import will fail for those records, but work for other records.

7
Integration with Microsoft Office

In this chapter, we will cover the following recipes:

- Configuring and using the Excel Data Connector add-in
- Using Workbook Designer
- Export API
- Lookup in Excel - creating a custom lookup
- Document management
- Creating a Word document with repeated elements

Introduction

In our day-to-day operations we use Microsoft Office a lot to store and retrieve numerical data in a grid format of columns and rows to calculating and analyze company data such as sales figures, sales taxes, or commissions. Dynamics 365 for Finance and Operations provides us with integration with Microsoft Office, which includes Microsoft Excel, Microsoft Word, the Document Management subsystem, and email. In this chapter, we will see how Microsoft Dynamics 365 for Finance and Operations integrates with Excel and Word by using data entities as an entry point into the system, how Excel can become a core part of the user experience, and how Excel and Word can be used for ad hoc lightweight reporting. We will also see how files can be stored and shared by using the Document Management and email capabilities in Dynamics 365 for Finance and Operations.

Configuring and using the Excel Data Connector add-in

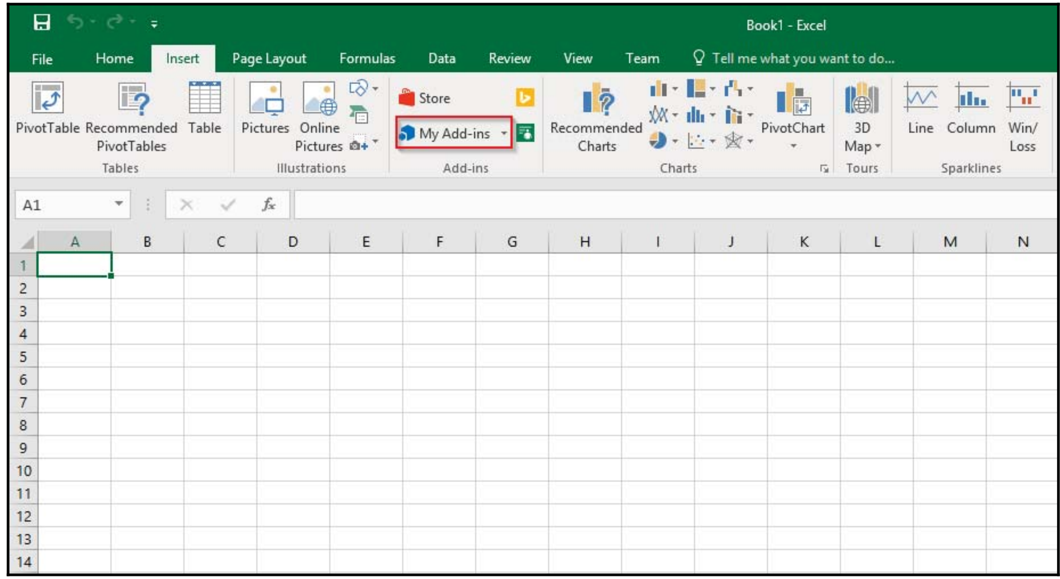
Microsoft Excel can change and quickly analyze data. The Excel Data Connector app interacts with Excel workbooks and OData services that are created for publicly exposed data entities. The Excel Data Connector add-in enables Excel to become a seamless part of the user experience. The Excel Data Connector add-in is built by using the Office Web Add-ins framework. The add-in runs in a task pane. Office Web Add-ins are web applications that run inside an embedded Internet Explorer browser window.

In this recipe, we will show you how to configure an Excel Data Connector add-in and use it to export data from Dynamics 365 for Finance and Operations and publish the updated data back in Dynamics 365 for Finance and Operations.

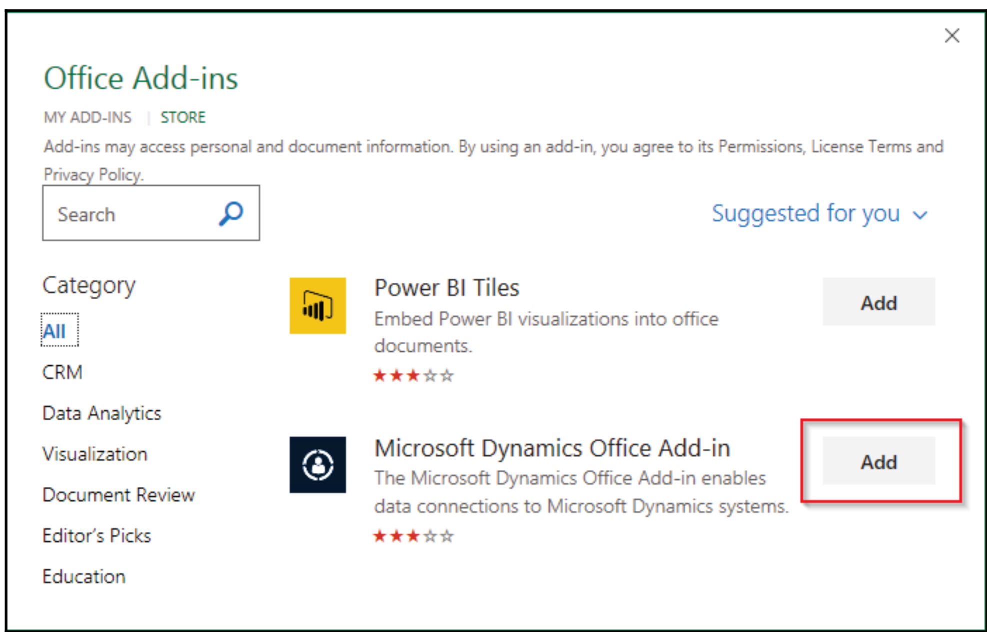
How to do it...

Carry out the following steps in order to complete this recipe:

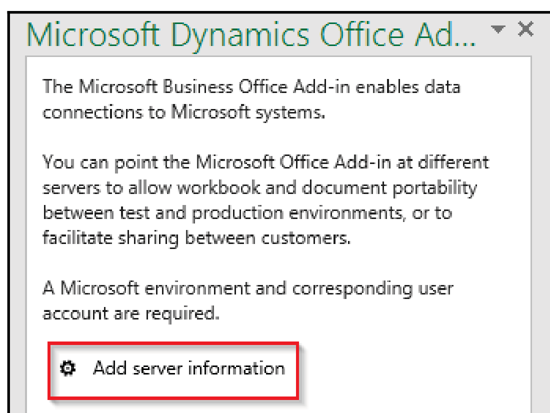
1. Open a new workbook in Excel and navigate to the **Insert** tab. Under **Add-ins**, select **My Add-ins**, as shown in the following screenshot:



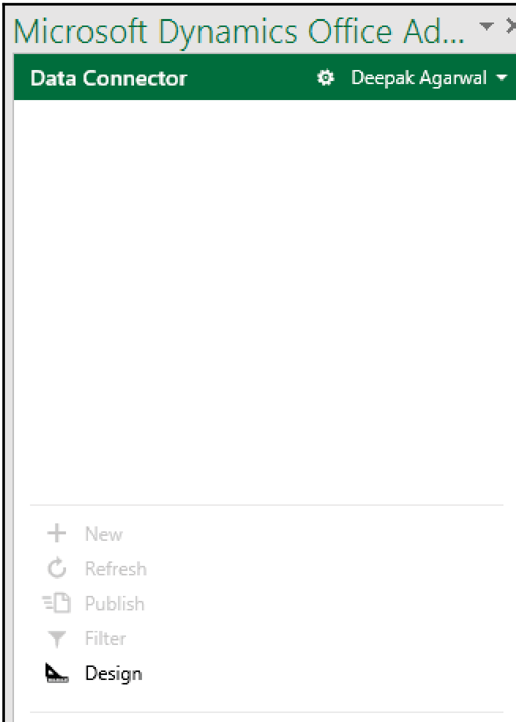
2. Under **Office Add-ins**, navigate to **STORE**, search for **Microsoft Dynamics Office Add-in**, and click on **Add**. This is shown in the following screenshot:



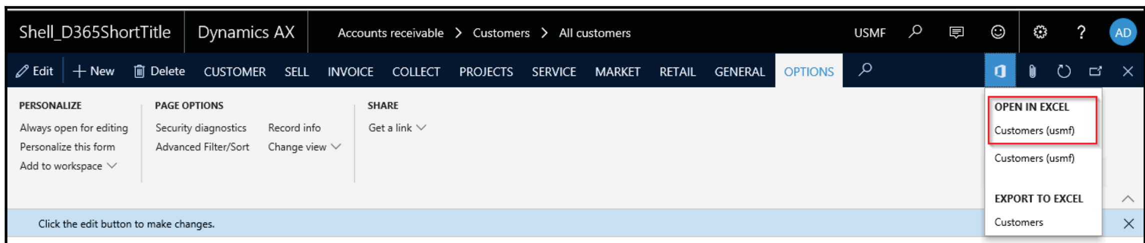
3. Once you've clicked, a new window will open within Excel as follows. Click on **Add server information** and the system will ask for account credentials. You have to sign in as the user that has access in your Dynamics 365 for Finance and Operations:



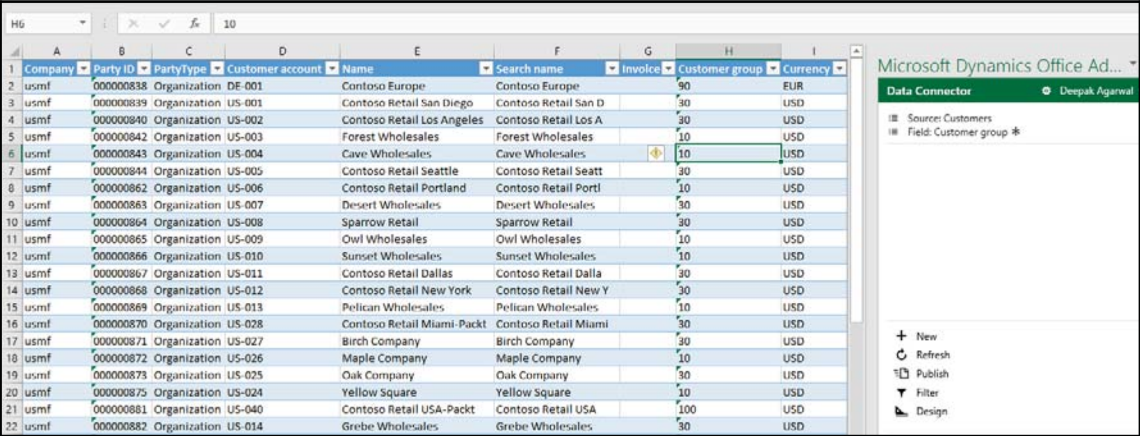
4. This will open up a Data Connector add-in app window as follows:



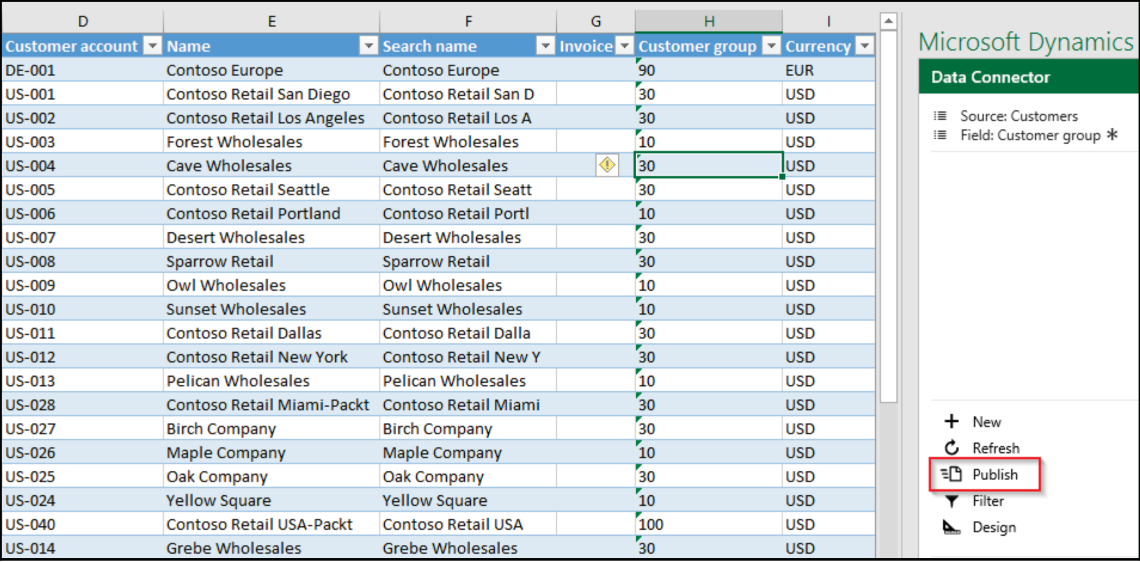
5. Let's navigate to **Accounts receivable** | **Customers** | **All Customers**, press the **Open in MS office** button, select **OPEN IN EXCEL**, and choose to **Download**:



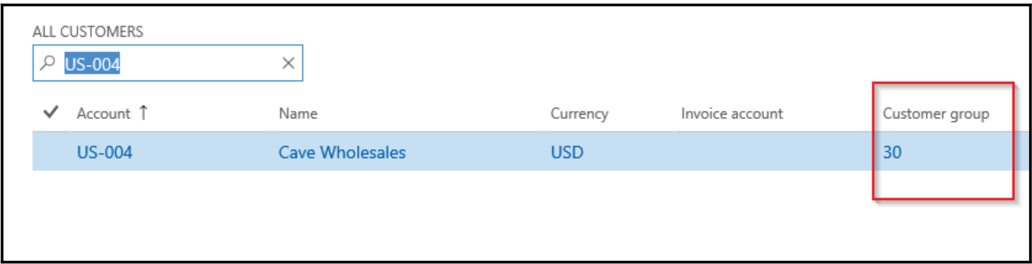
6. Let the file reload and retrieve the data. It will open the file as follows. Select a customer record to update the **Customer group** field to change its value from `10` to `30`:



7. Once the **Customer group** value has been changed, then select the **Publish** button to update the record in Dynamics 365 for Finance and Operations. Consider the following screenshot:



8. Filter the customer in the **ALL CUSTOMERS** form and check the result, as shown in the following screenshot:



How it works...

We start the recipe by first installing the Microsoft Dynamics Office add-ins for Dynamics 365 for Finance and Operations. Data from Dynamics is exposed to the external world by using OData entities, and authentication is done via Azure Active Directory, providing that the user signed in has access to Office 365. OData sits on the same authentication stack as the server. The add-in uses OAuth to facilitate authentication.

We can use the Microsoft Dynamics Excel Data Connector App (Excel App) to create, read, update, and delete Dynamics 365 for Finance and Operations. The connector uses OData services that are created for any entity that is left in the default state of Public (**DataEntity.Public**=`Yes`).

The following text will suggest how the data is accessed by Excel by using various technologies:

Excel | Office Web Add-in (JS + HTML) | JavaScript OData API (Olingo) | Authentication through Azure Active Directory (AAD) | Dynamics 365 for Finance and Operations OData services on the AOS | Dynamics 365 for Finance and Operations Entities | Dynamics 365 for Finance and Operations LINQ provider | Dynamics 365 for Finance and Operations Database.

Using Workbook Designer

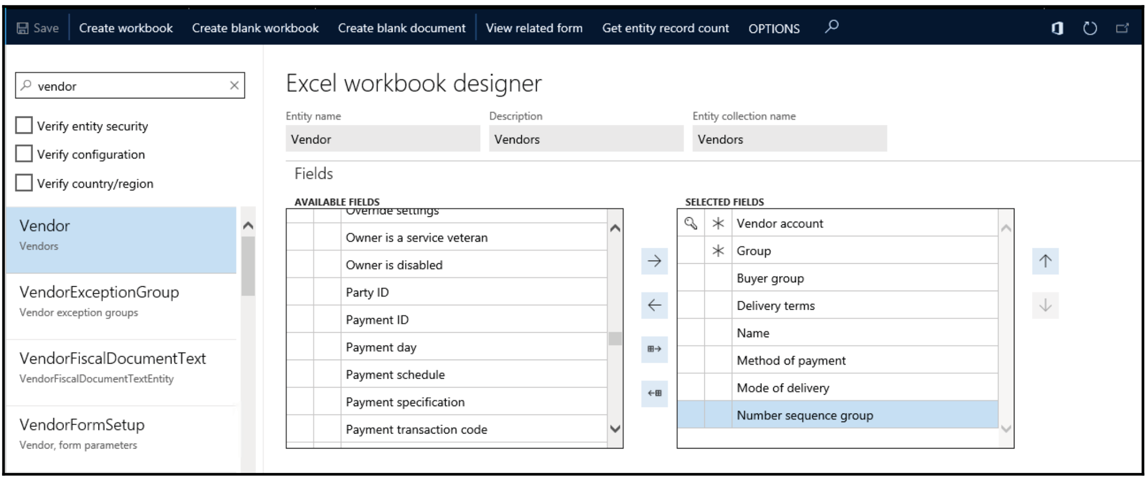
We can use the Workbook Designer page to design an editable custom export workbook that contains an entity and a set of fields. Before we can publish data edits, all the key fields of the entity must be in the Excel table. Key fields have a key symbol next to them. To successfully create or update a record, it must have all the mandatory fields in the Excel table. Mandatory fields have an asterisk (*) next to them.

In this recipe, we will show you how to design an Excel workbook from Dynamics 365 for Finance and Operations and use it to export data from Dynamics 365 for Finance and Operations and publish the updated data back in Dynamics 365 for Finance and Operations. We could also import the same workbook to use it as a document template.

How to do it...

Carry out the following steps in order to complete this recipe:

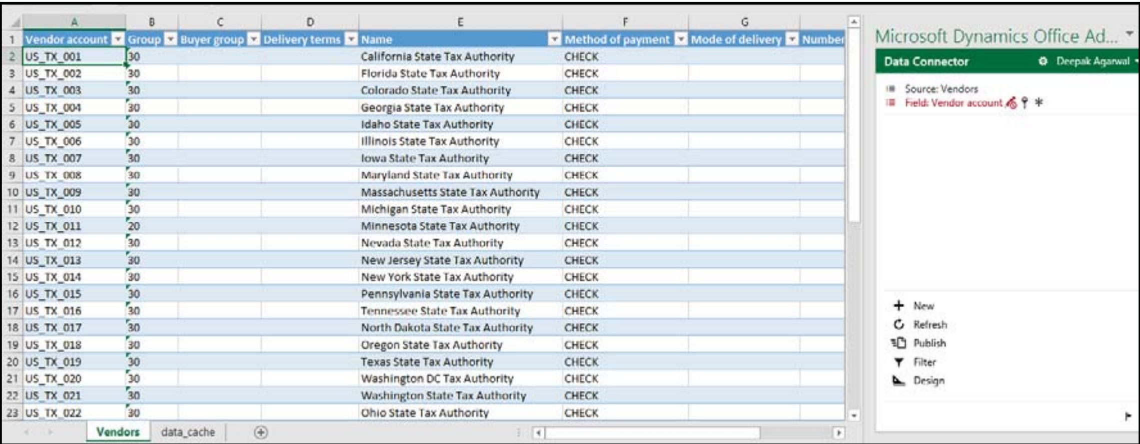
1. Navigate to **Common** | **Common** | **Office Integration** | **Excel workbook designer** form in Dynamics 365 for Finance and Operations.
2. Find the data entity **Vendor** and select the fields from the available fields, as follows:



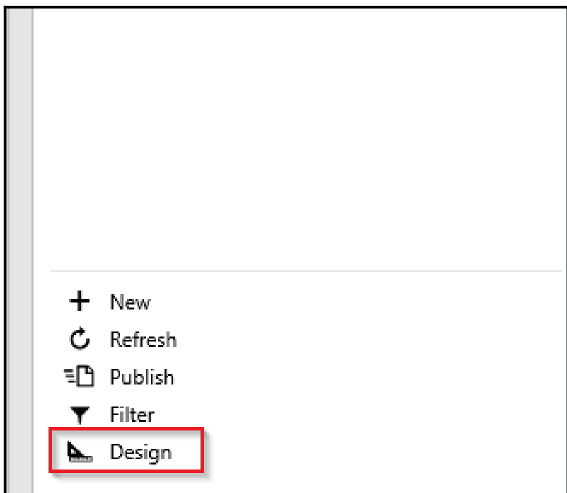
3. Click the **Create workbook** button. The **Create workbook** button will add the selected entity and fields, a pointer to the server, and the app into a workbook.

4. The **Get entity record count** button will show the record count for the currently selected entity. Currently, the Excel Data Connector App cannot handle large (tall and wide) data-sets. Any unfiltered entity with more than `10,000` records is at risk of crashing the app.

5. After creating the workbook, enable editing in Excel and the data will be loaded as follows:

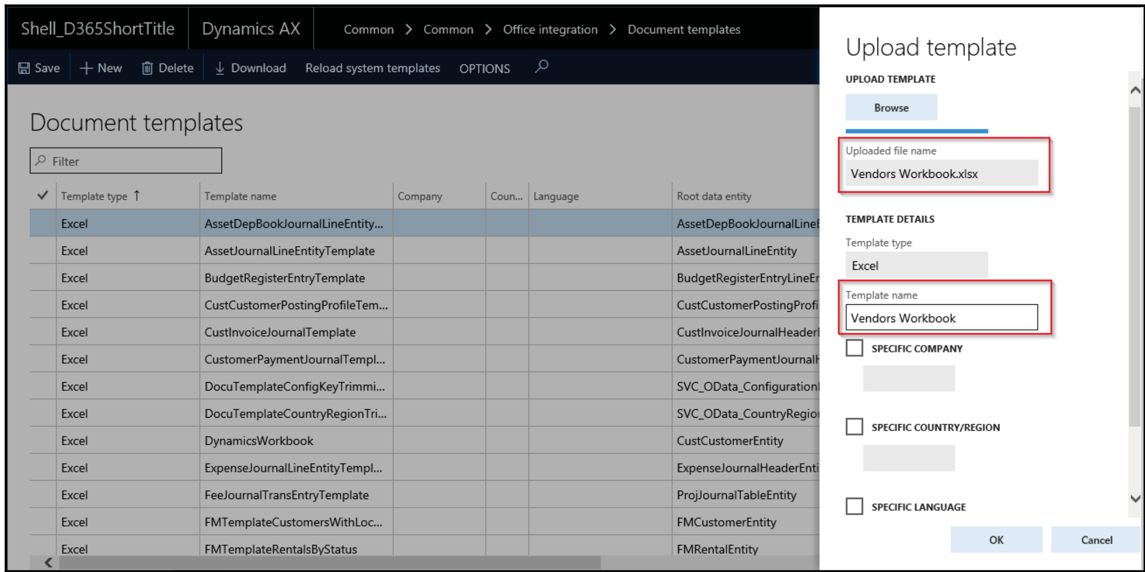


6. After obtaining a workbook with an Excel Data Connector App, additional data sources can be added using the **Design** button:

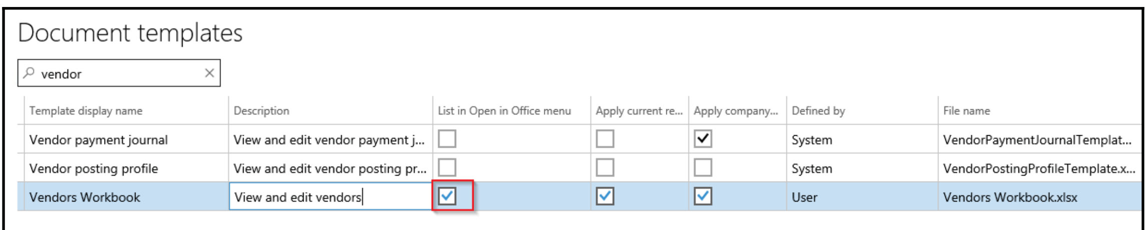


7. We could edit data here and add this workbook to **Document Templates** in Dynamics 365 for Finance and Operations, so that it will be added as an option in the **Open in Excel** section of the **Open in Microsoft Office** menu.

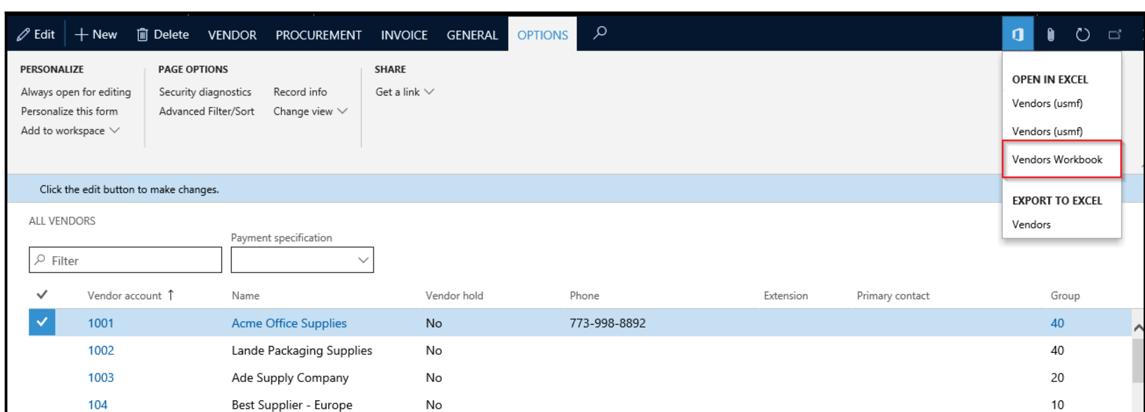
8. Navigate to **Common** | **Common** | **Office Integration** | **Document Templates**:



9. Note here, **List in Open Office Menu** will be checked for the record created for our custom **Vendors Workbook** record:



10. Navigate to **Accounts payable** | **Vendors** | **All Vendors**:



How it works...

Dynamics 365 for Finance and Operations has an inbuilt framework to design workbooks, where the user is presented with the Data entities dropdown. When we select data entity fields for export, to retrieve the resulting workbook, click **Create Workbook** in the app bar. We can save this workbook and it can be imported as a Word document template. Once this is imported into the system and has a **List in Open Office** menu, then it is automatically added as a generated Excel template.

Export API

The Export API is used to provide custom export options such as **Open in Excel** options that are manually added via the Export API.

In this recipe, we will show how to use the Export API to provide custom export options by adding an explicit button for **Open in Excel** experiences. The label shown on the button should usually be **Open target in Excel** where the target is the name of the target data, such as `Vendors` or `catalog`. The code behind such a button will be for obtaining the template to be used. Add the desired filter and pass the template to the user.

How to do it...

Carry out the following steps in order to complete this recipe:

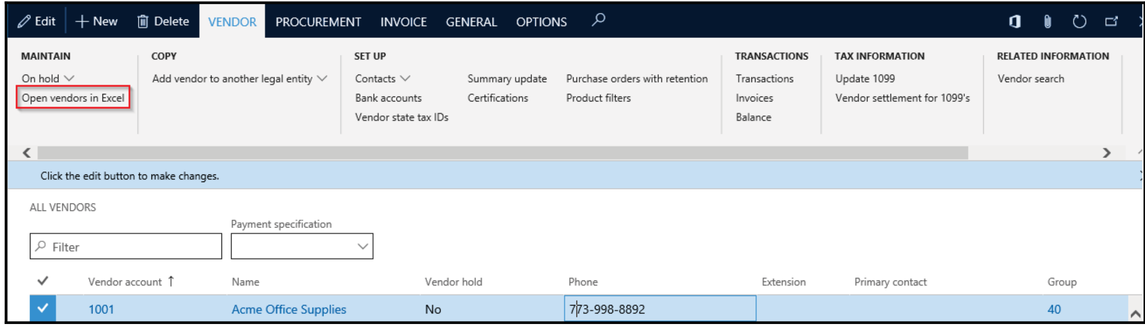
1. Start Visual Studio 2015 by opening the previously created project, or create a new project.
2. Right-click on the project, and then click **Add** | **New item**.
3. Select the **Resource item type** and set the name to `VendorBasicTemplate`. Make sure that `VendorBasicTemplate.xlsx` is closed.
4. Add a document template for usage in code.

5. Add a new button `OpenVendorInExcel` on the `VendTable` form under the path `\Forms\VendTable\Design\ActionPane(ActionPane)\VendorTab(ActionPaneTab)\VendorModify(ButtonGroup)\OpenVendorInExcel(Button)`.
6. Add code on the `clicked` method on the **OpenVendorInExcel** button:

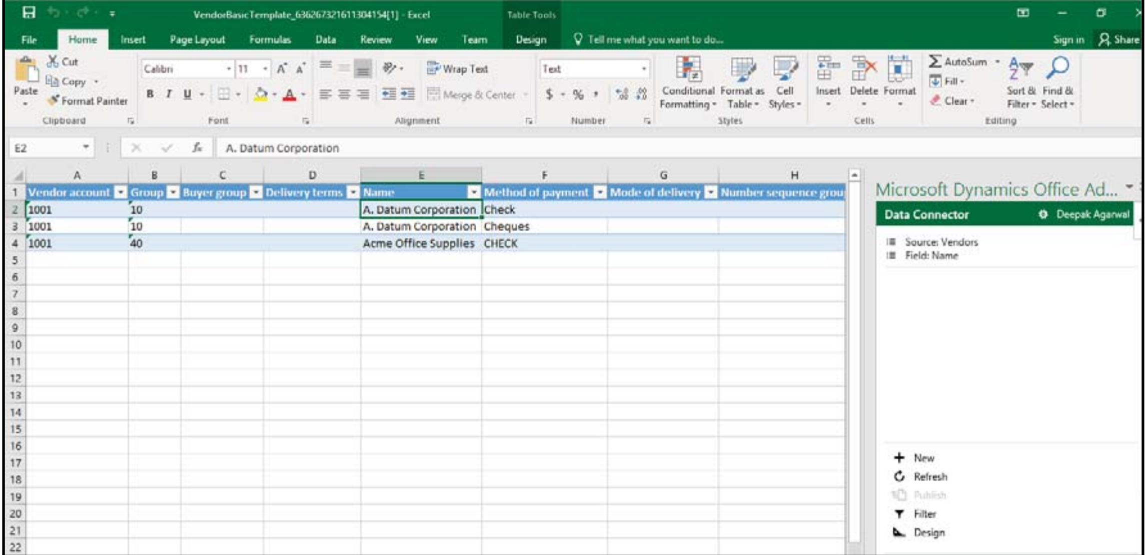
```
[Control("Button")]
class OpenVendorInExcel
{
  public void clicked()
  {
    super();
    const str templateName = resourceStr(VendorBasicTemplate);
    DocuTemplate template =
      DocuTemplate::findTemplate(OfficeAppApplicationType::Excel,
       templateName);
    // Ensure the template was present
    if (template && template.TemplateID == templateName)
    {
      Map filtersToApply = new Map(Types::String, Types::Class);
      // Create vendors filter
      ExportToExcelFilterTreeBuilder filterBuilder = new
      ExportToExcelFilterTreeBuilder(tablestr(VendVendorEntity));
      anytype filterString =
      filterBuilder.areEqual(fieldstr(VendVendorEntity,
       VendorAccountNumber), VendTable.AccountNum);
```

```
            filtersToApply.insert(tablestr(VendVendorEntity),
            filterString);

            // generate the workbook using the template and filters
            DocuTemplateRender renderer = new DocuTemplateRender();
            str documentUrl = renderer.renderTemplateToStorage(template,
              filtersToApply);
            // Pass the workbook to the user
            if (documentUrl)
              {
                Browser b = new Browser();
                b.navigate(documentUrl, false, false);
              }
              else
              {
                error(strFmt("@ApplicationFoundation:DocuTemplate
                  GenerationFailed", templateName));
              }
          }
          else
          {
           warning(strFmt("@ApplicationFoundation:DocuTemplateNotFound",
               templateName));
          }
        }
      }
```

7. Navigate to **Accounts payable** | **Vendors** | **All Vendors** and find the button **Open Vendors in Excel**:



8. This button will export the selected record. Here, we have applied filters on the vendor code so it will only export the vendor code `1001` record from all the companies. To select a record from a particular company, apply filters on `dataAreaId`:



How it works...

Here, we first need to add the resource which refers are Excel template. This same template should be loaded as a document template in Dynamics 365 for Finance and Operations. Once this is done, we can use the same template to export the data to Excel in our code.

We can add a filter while retrieving data by using `ExportToExcelFilterTreeBuilder`, which returns a filter expression of class type. Then, insert these values in `Map` type and use the `DocuTemplateRender` class to render the document from the method `renderTemplateToStorage()`, which returns the browser URL. The browser URL can be passed to the `navigate ()` method of the `Browser` class.

Lookup in Excel - creating a custom lookup

To facilitate data entry, the Excel App provides lookups and data assistance. Date fields provide a date picker, enumeration (enum) fields provide an enum list, and relationships provide a relationship lookup. When relationships exist between entities, a relationship lookup is shown. We can create custom lookups to show data options when an enum or relationship isn't sufficient. The main use case is when data must be retrieved from an external service and presented in real time.

In this recipe, we will show how to add a custom lookup on currency code on a vendor's entity.

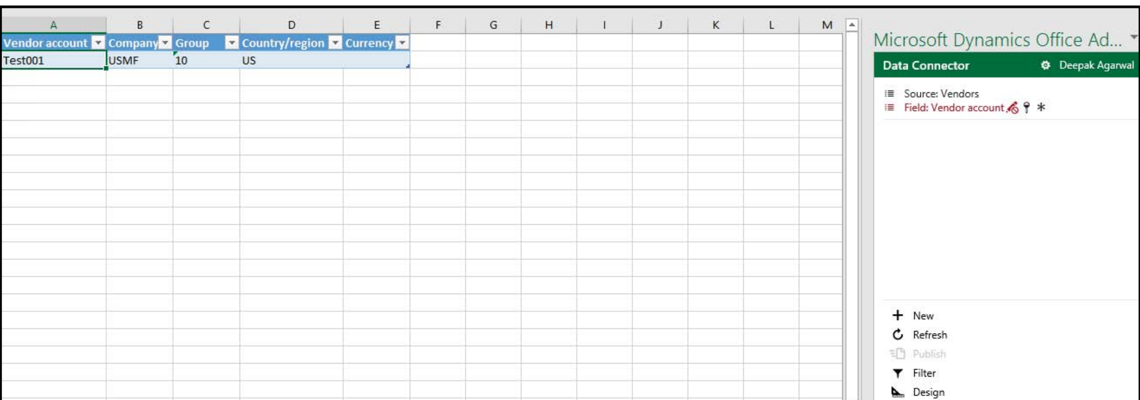
How to do it...

Carry out the following steps in order to complete this recipe:

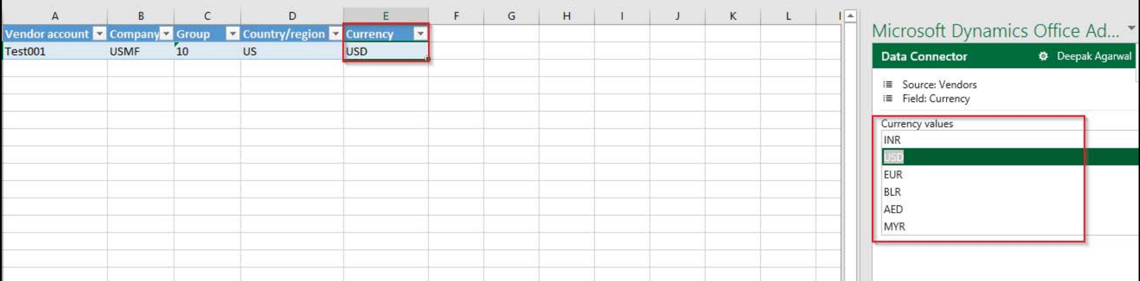
1. Open the previously created project in Visual Studio.
2. Open the Designer View for **VendVendorEntity**. Right-click on **Methods** and then click on **New Method**.
3. Add the `lookup_CurrencyCode` method from the following code sample:

```
[SysODataActionAttribute("VendVendorEntityCustomLookup", false),
//Name in $metadata
SysODataCollectionAttribute("_fields", Types::String),
//Types in context
SysODataFieldLookupAttribute("CurrencyCode")] //Name of field
public static str lookup_CurrencyCode(Array _fields)
{
  OfficeAppCustomLookupListResult result =
    new OfficeAppCustomLookupListResult();

  result.items().value(1, "INR");
  result.items().value(2, "USD");
  result.items().value(3, "EUR");
  result.items().value(4, "BLR");
  result.items().value(5, "AED");
  result.items().value(6, "MYR");
  return result.serialize();
}
```

4. Open **Excel Workbook**, and open designer to add fields as follows:



5. Select the **Currency** Code (**Currency** as shown in the following screenshot) column; have a look at the **Add-ins** window for lookup values:



How it works...

Here, Dynamics 365 for Finance and Operations Data entity API automatically recognizes `SysODataActionAttribute`, `SysODataCollectionAttribute`, and `SysODataFieldLookupAttribute`. Also, we need to add the method name as `lookup_fieldname()` so that the system considers it to render the lookup for the field. Here, in this example, we have provided lookup on **CurrencyCode**. When we navigate to the field, the system uses OData API and retrieves the lookup values.

Document management

In Dynamics 365 for Finance and Operations, document management supports Azure Blob and SharePoint online for saving record attachments. Database storage is not supported anymore.

Azure Blob: Azure Blob storage is the default storage for Dynamics 365 for Finance and Operations. It is equivalent to storage in a database, as documents can only be accessed through Dynamics 365 for Finance and Operations. Also, it provides the added benefit of providing storage that doesn't negatively affect the performance of the database.

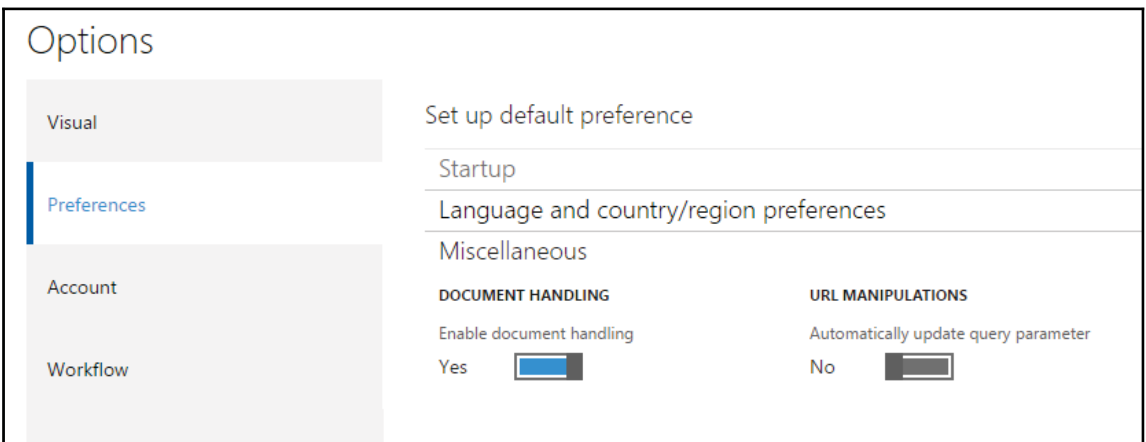
SharePoint Online: When you have an O365 license and MS, autodiscover the SharePoint tenant, for example, a user on the `packtPublication.onmicrosoft.com O365/AAD` tenant gets `packtPublication.sharepoint.com` as the SharePoint site. This whole architecture allows SharePoint storage to work immediately.

How to do it...

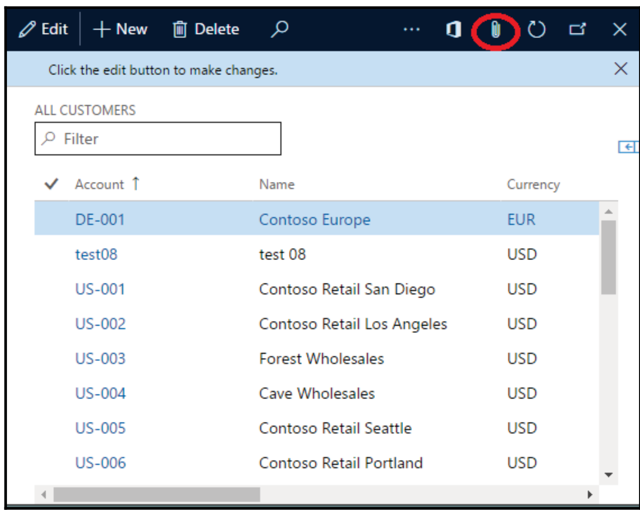
Carry out the following steps to understand document management:

1. As the very first step, check whether **DOCUMENT HANDLING** is enabled or not. You can turn on this feature through the **Options** button in the right top corner on your screen by following this path:

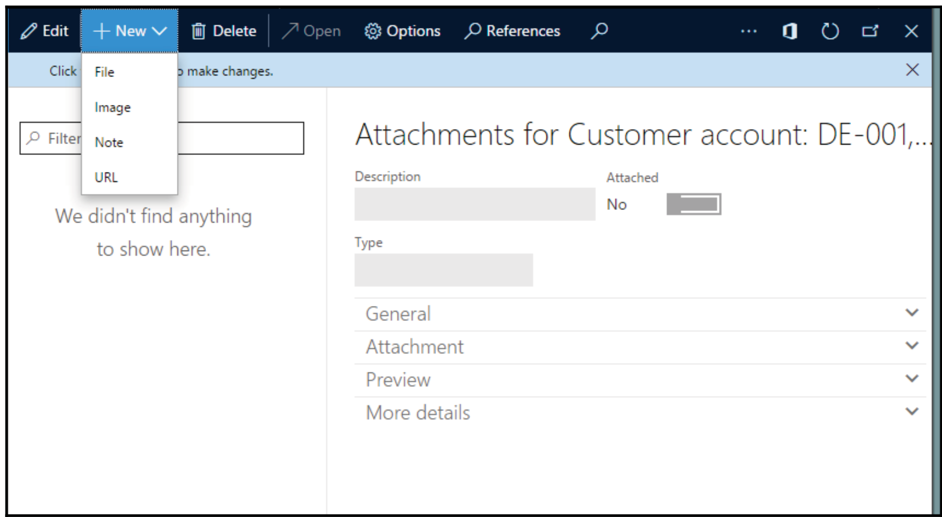
   Options | **Preferences/General** | **Miscellaneous** and set the document handling as active:



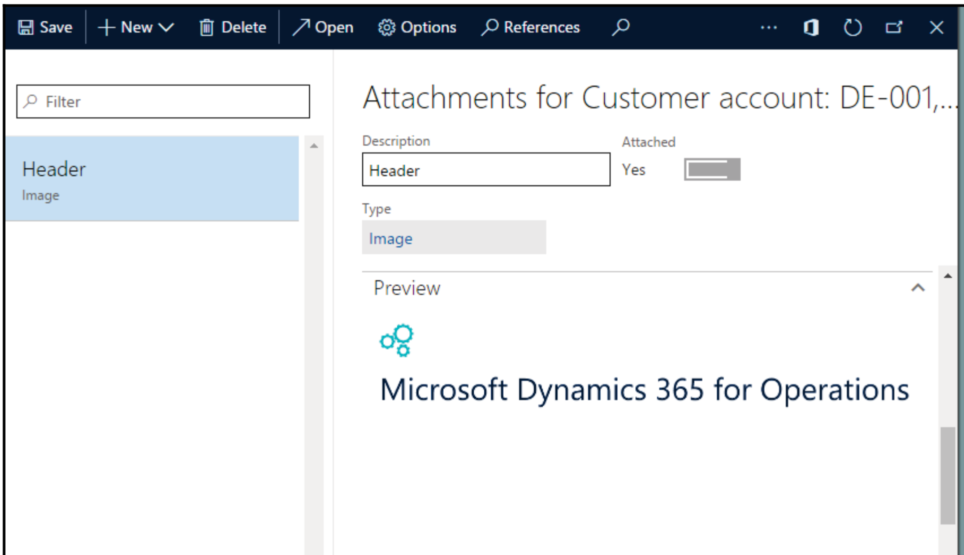
2. Once this feature is enabled, you will get the attach button in the upper-right corner, as shown in the following screenshot. To check, open any form (**Account receivable** | **Customers** | **All customers**):



3. Now, to add any document, click on the **New** button. You will have an option to choose from **File**, **Image**, **Note**, and **URL** document types and the system will ask you to upload the attachment:



4. Let's try to attach an image file. Select **Image** from the **New** dropdown. Select an image file. Once the upload is done, you can check the related information on the form along with the preview:



5. On this form, you will see some more fast tabs that contain more details about this attachment.

How it works...

In Dynamics 365 for Finance and Operations, when you select an attachment, it will store it on your selected storage, while using the Azure storage system will create a Blob and link it to the current record.

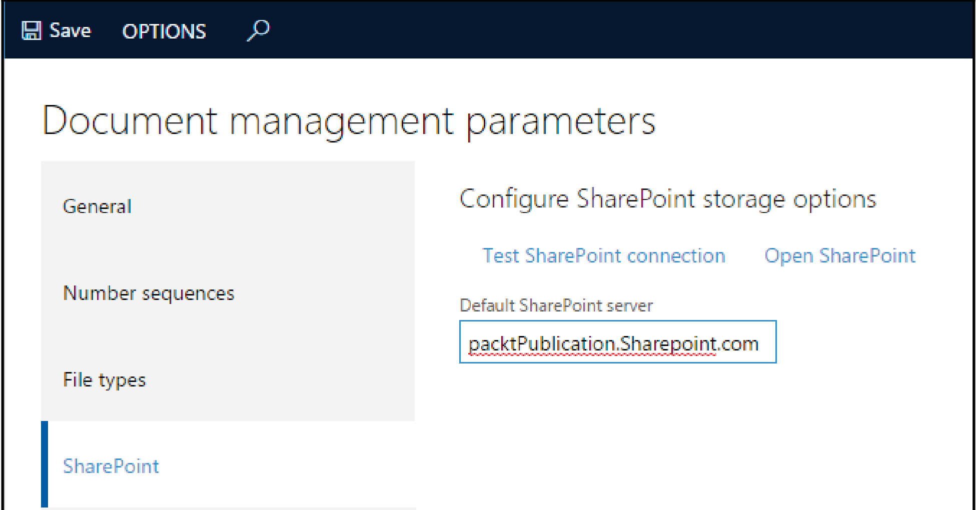
There's more...

You may have found some changes in different updates of Dynamics 365 for Finance and Operations, as earlier it was Microsoft Dynamics 365 for Operation and still some more exciting updates are in the queue.

For example, in the preceding recipe, we showed four document types, while in current versions you may get **File** and **Image** only.

You can also attach Office document types, such as Microsoft Word, Excel, and PowerPoint files, but for that you must use a production Office Web Apps Server, which might not be available in a OneBox configuration.

Once you have an Office365 license, you need to set up SharePoint storage for attachments. To do that, go to **Organization administration** | **Document management** | **Document management parameters** and ensure that the SharePoint server has been automatically discovered and set:



Test this connection and, on successful setup, open or create a Document type, set the document type's location to `SharePoint`, and select the folder that the files should be stored in.

8
Integration with Power BI

In this chapter, we will cover the following recipes:

- Configuring Power BI
- Consuming data in Excel
- Integrating Excel with Power BI
- Developing interactive dashboards
- Embedding Power BI visuals

Introduction

Power BI is best known as a reporting tool that helps you to create interactive visual reports. When it comes to Dynamics 365 for Finance and Operations, now it's a cloud-based analytics visualization platform that helps you to extract, transform, and present business data from Microsoft Dynamics 365 for Finance and Operations to interactive reports and dashboards.

This chapter explains the out-of-box integration between Dynamics 365 for Finance and Operations and `PowerBI.com`. You will also learn how we can use the features and its services that are part of Microsoft Power BI to access, explore, and gain insight from your Microsoft Dynamics 365 for Finance and Operations.

Power BI provides you a self-service analytics solution while working with the Office 365 cloud service and automatically refreshes the Microsoft Dynamics 365 for Finance and Operations data into your BI dashboards. You can also use Power BI Desktop or Microsoft Office Excel **Power Query** for authoring reports and Power BI for sharing dashboards and refreshing data from Microsoft Dynamics 365 for Finance and Operations, this allows your organization to use a powerful way to work with Dynamics 365 data.

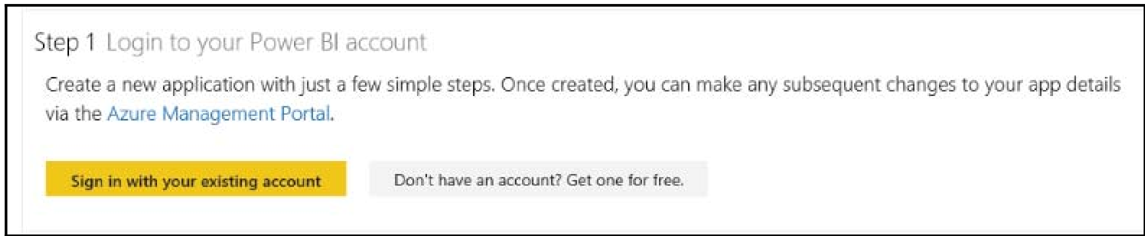
Configuring Power BI

You can connect to Microsoft Dynamics 365 for Finance and Operations with Power BI Desktop to create custom Dynamics 365 reports and dashboards for use with the Power BI service. To enable Power BI the very first thing you have to do is configure your Power BI with Dynamics 365 for first use.

How to do it...

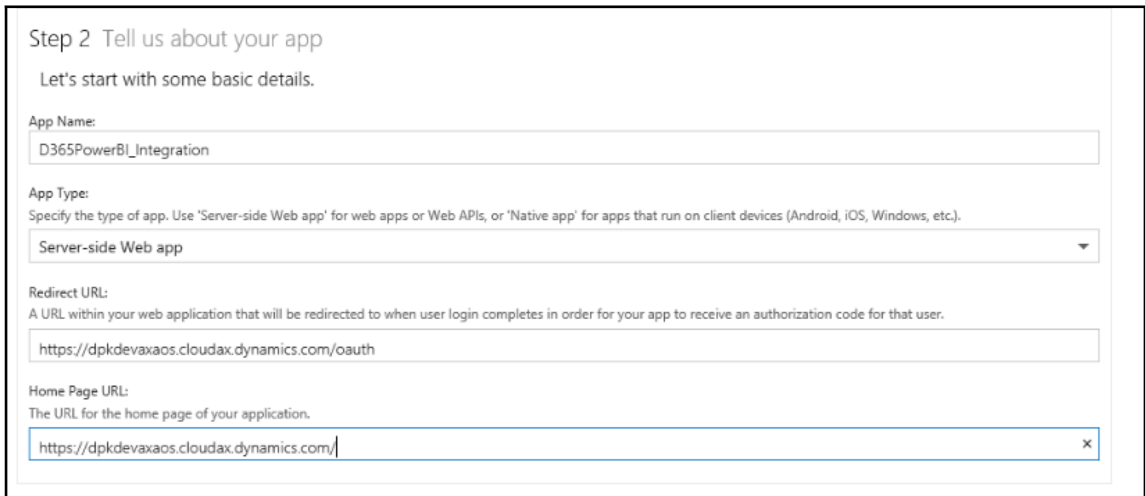
Follow these steps:

1. Log in to your Power BI account using `https://powerbi.microsoft.com/en-us/`.

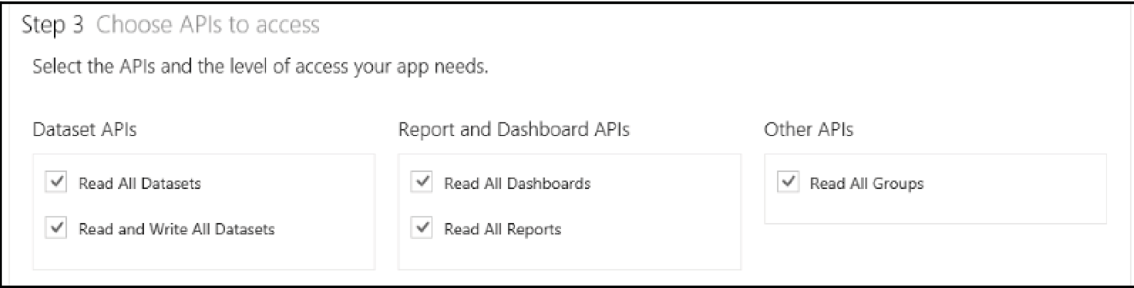


2. Give details about the application and Dynamics 365 for Finance and Operations environment, you can refer to the following for more clarity:

- **App Name**: Give a generic application name
- **App Type**: Choose Server side web app
- **Redirect URL**: Add oauth at the end of your home page URL
- **Home page URL**: Application URL

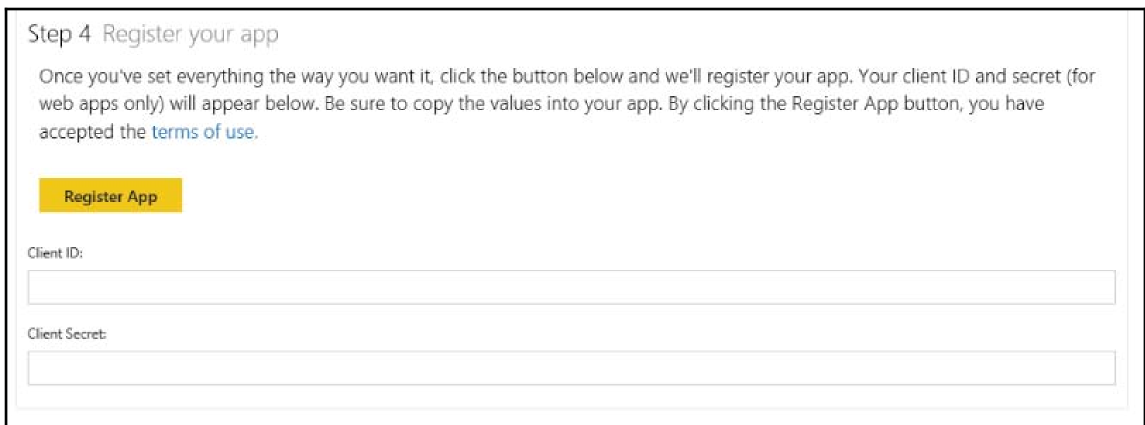


3. Choose APIs to access:



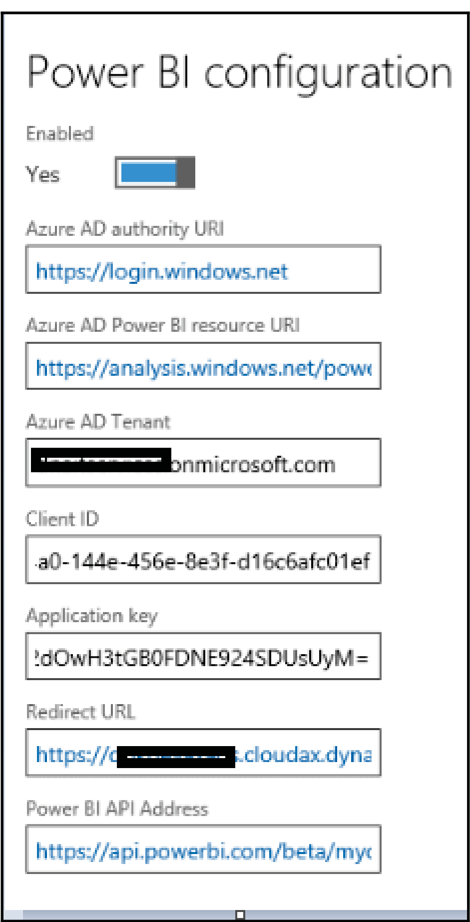
4. Register your app.

   In this step, click on **Register App** and the system will create `client ID` and `client secret` for your new app. Keep a backup of both values, you need to put these values in the next step:

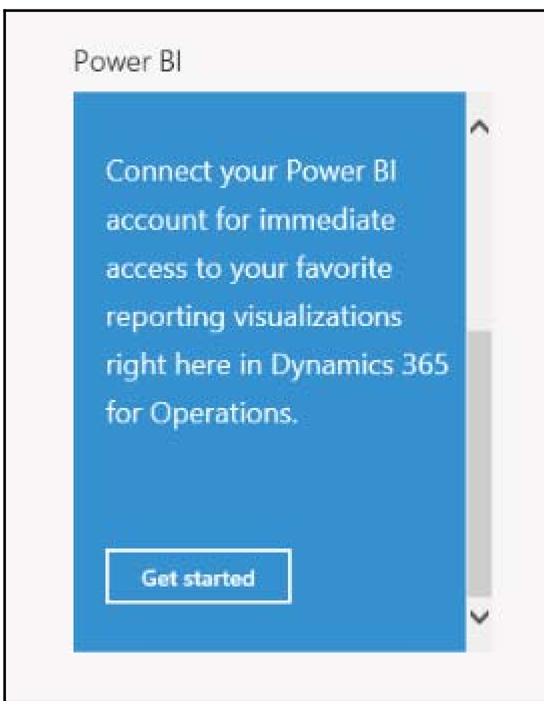


5. Now open Dynamics 365 for operation and navigate to **System Administration** | **Setup** | **Power BI.**

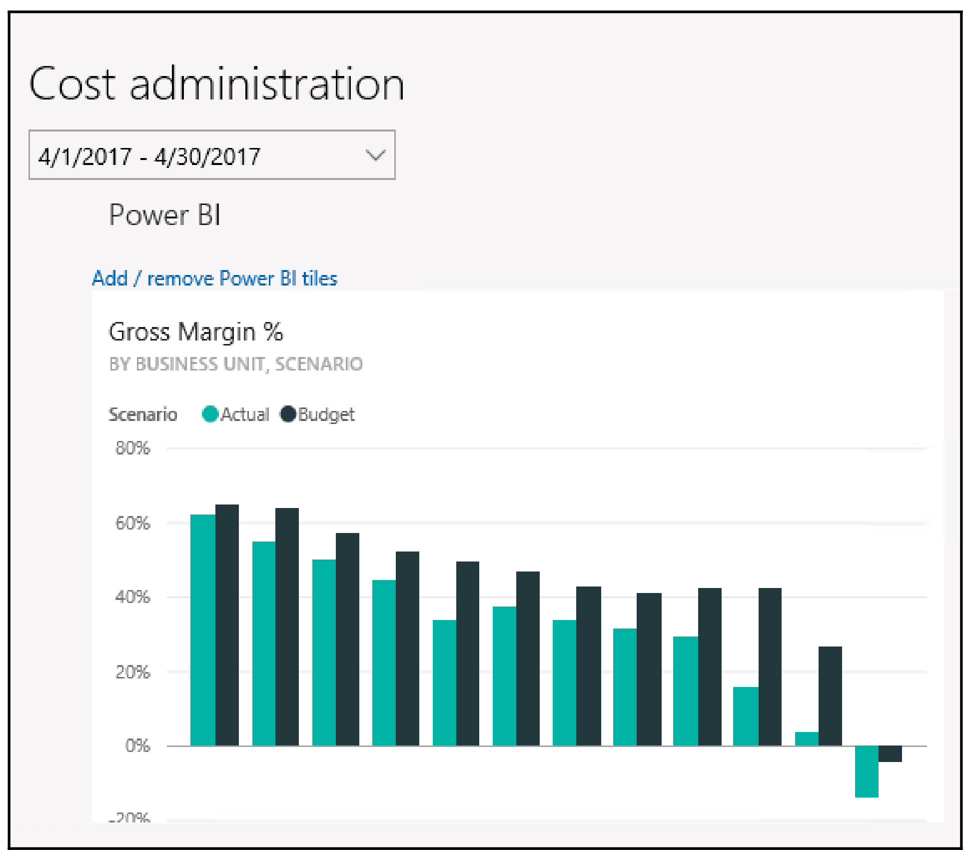
Use `client ID` and `Application Key`, and add the details here which has been generated from the last step respectively:



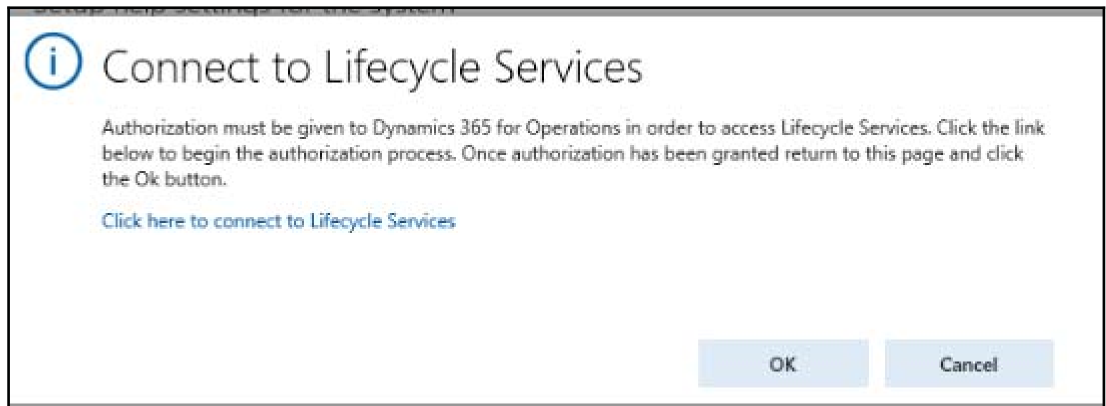
6. Now open any workspace, on the right side you will get a **Power BI** tile, click on **Get started**:



7. Now authorize Power BI using admin credentials. Once done with this step you will get power BI tiles on the workspace:

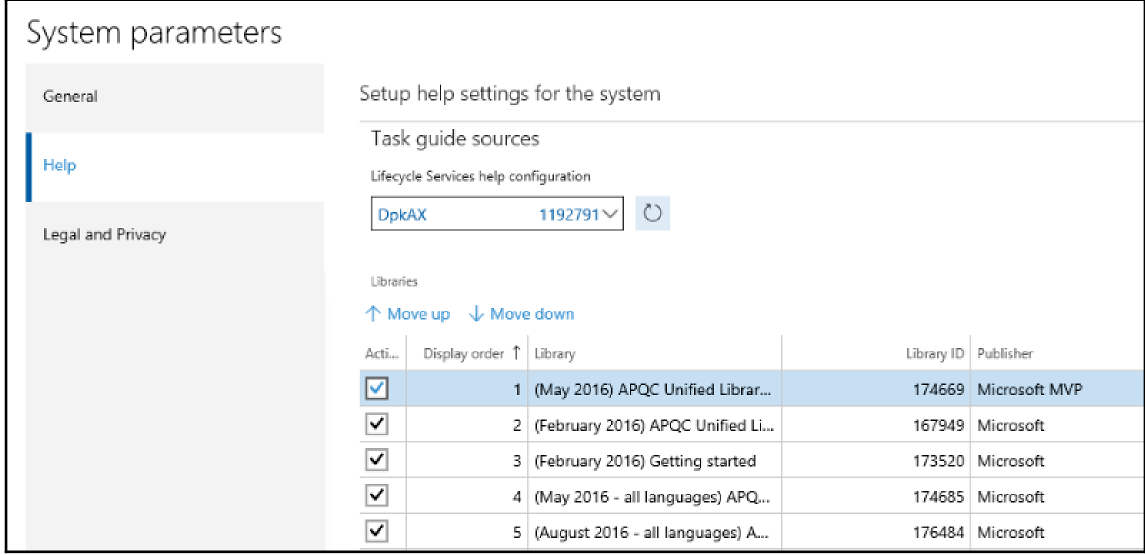


8. Now navigate to **System administration** | **Setup** | **System parameter**, and go to the **help** tab. You will get a notification for connecting with LCS, select **Click here to connect to LCS**:

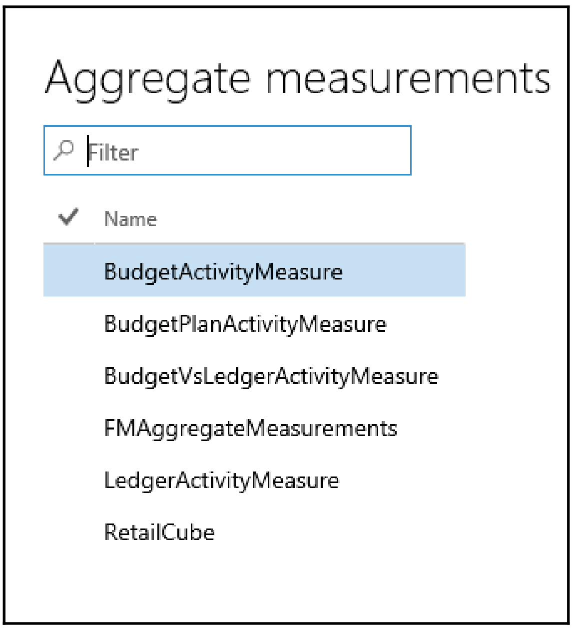


On successful authentication, you will get selection data in the **Help** tab.

9. Fill in the required details here:



10. Now navigate to **System administrator** | **Setup** | **Entity Store** and refresh all `aggregate measurement`. You can set batch jobs as well for this activity:



11. Now you will get these in your LCS account under **Shred Asset library** | **Power BI reports**. You can download the `pbxi` files to modify the Power BI desktop application.
12. Go to **System administrator** | **Setup** | **Deploy Power BI files** and deploy all required BI files from this screen.

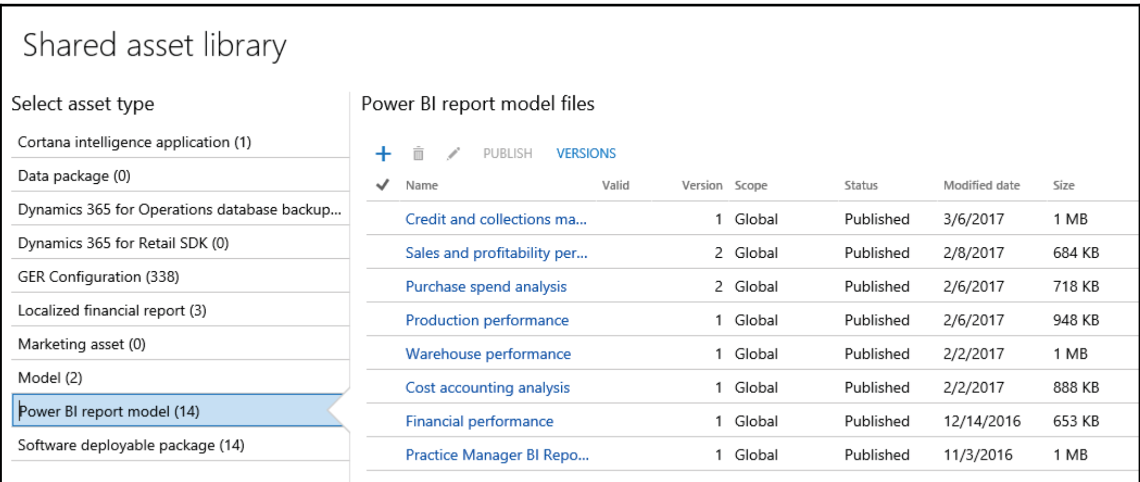
How it works...

When you log in to the Power Bi account using your organization credentials you will be able to access it. Using these steps, you can generate `Client Id` and `application id` for your power BI application. Use these details in Dynamics 365 to create mapping with a Power BI account.

After this initial mapping, you have to do more setup in Dynamics 365, as shown in the preceding recipe. Once you are done with all these steps you will be able to access Power BI tiles under each workspace.

There's more...

You can download a default Power BI report using LCS. Log in to your LCS account and select **Shared asset library**, under Power BI report Model you will see 14 default BI report model files:



Click any of them and you can download it on your local drive. Use Power BI desktop to modify these reports.

See also

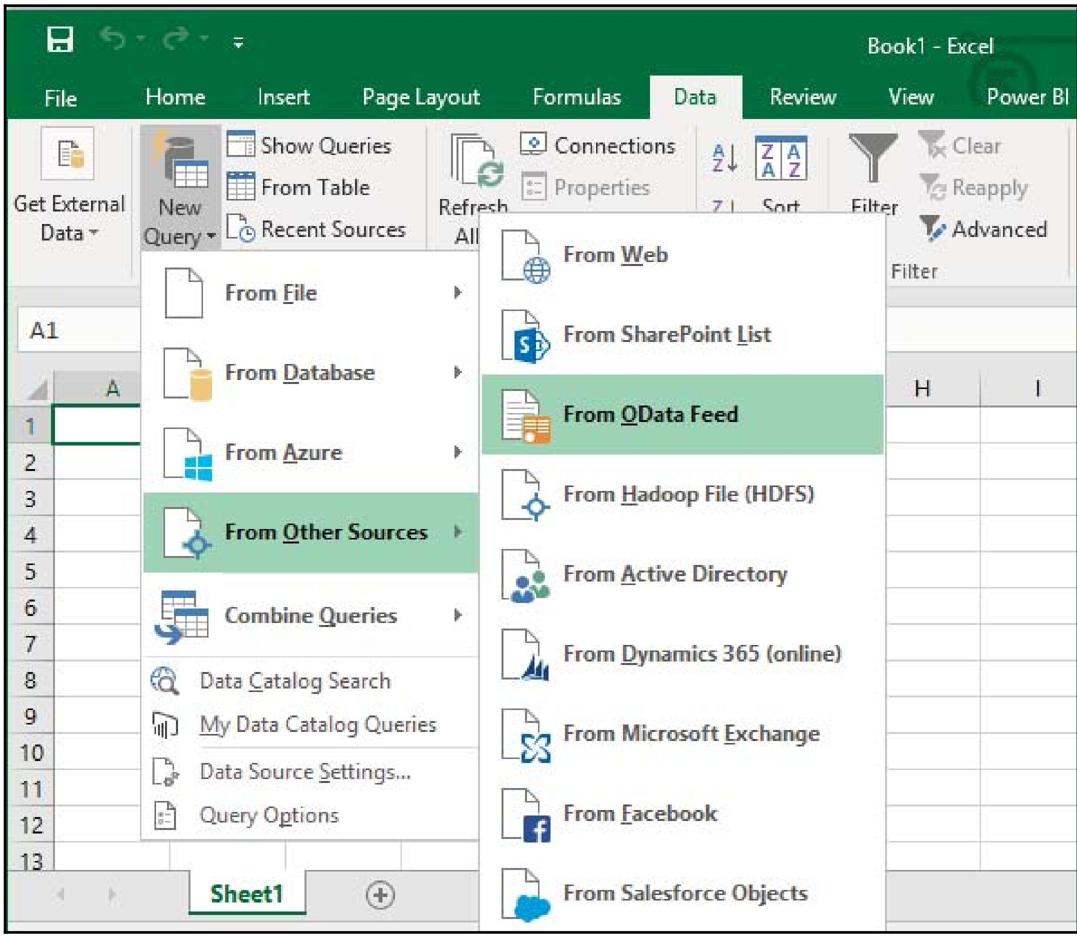
- You may need to install Power BI gateway to connect with the application, download this using the following link:
  `https://powerbi.microsoft.com/en-us/gateway/`

Consuming data in Excel

In this recipe, we will consume Dynamics 365 data into an Excel file using *power view* and *power pivot* tools. Follow with use of power view report. It's a very useful tool for reports and visual artifacts. There are many ways to import data into Excel, here in this recipe we will use the `OData` feed to get required data and convert it into visual reports.

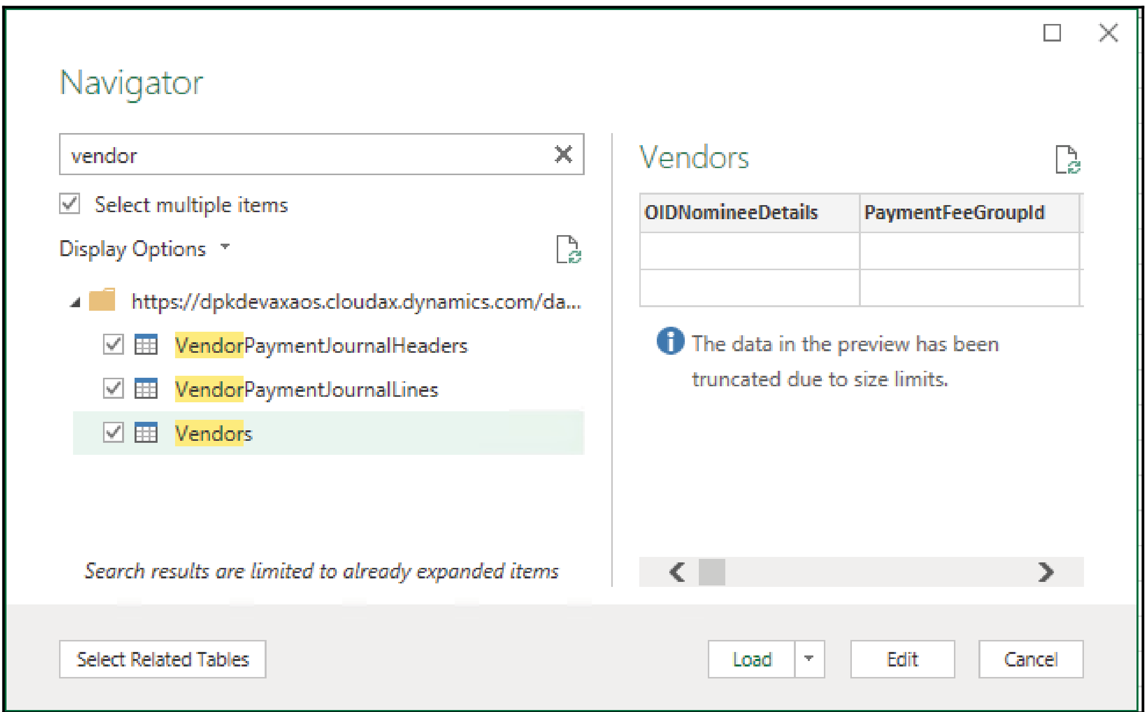
How to do it...

1. Open Excel and go to the **Data** tab. Click on **New Query** | **From other source** | **From OData feed**:



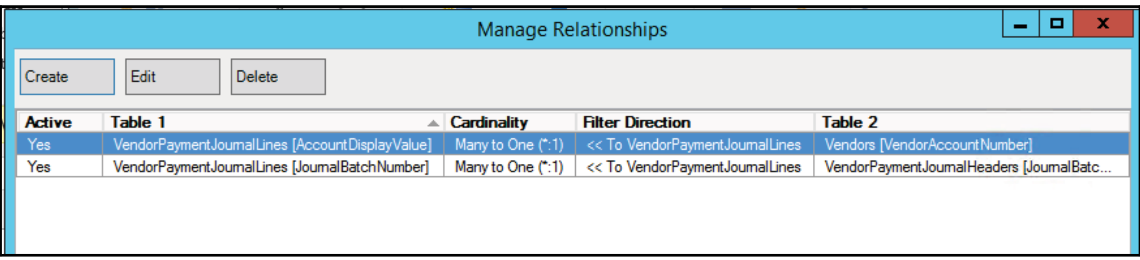
2. In the next screen add your Dynamics 365 API URL, add `/data` at the end of the URL, and click **OK**:

3. It will navigate you to the next screen where we need to select three tables, as shown in the following screenshot:



When you click the **Load** button, all queries will load on your Excel file right away. You can select every query for an update such as hiding/selecting the require column.

4. Now go to **Data tab** | **Data tool group** | **Manage Data Model**. It will open another screen for Power Pivot for Excel. Go to **Design Tab** | **Relationship group** | **Manage Relationships**, use this form to create/edit appropriate relationships between tables as follows:



Close the **Manage Relationships** dialog box.

5. Now try to add some calculated measures on the report, to do that, use the **Measures pane** at the bottom of the **PowerPivot Window** to add two new measures:

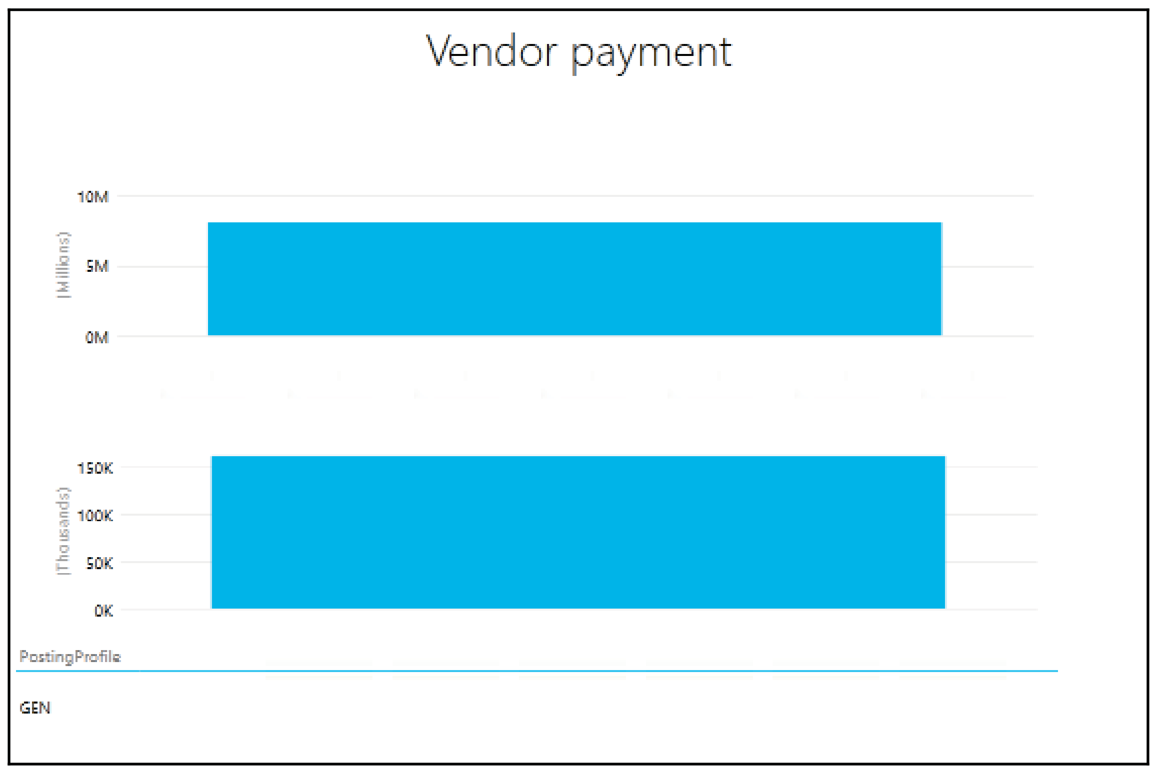
- Click on the **Measure** pane below your grid and enter the following formula, and then press *Enter*:

```
Total Debit:=CALCULATE(SUM([DebitAmount]))
```

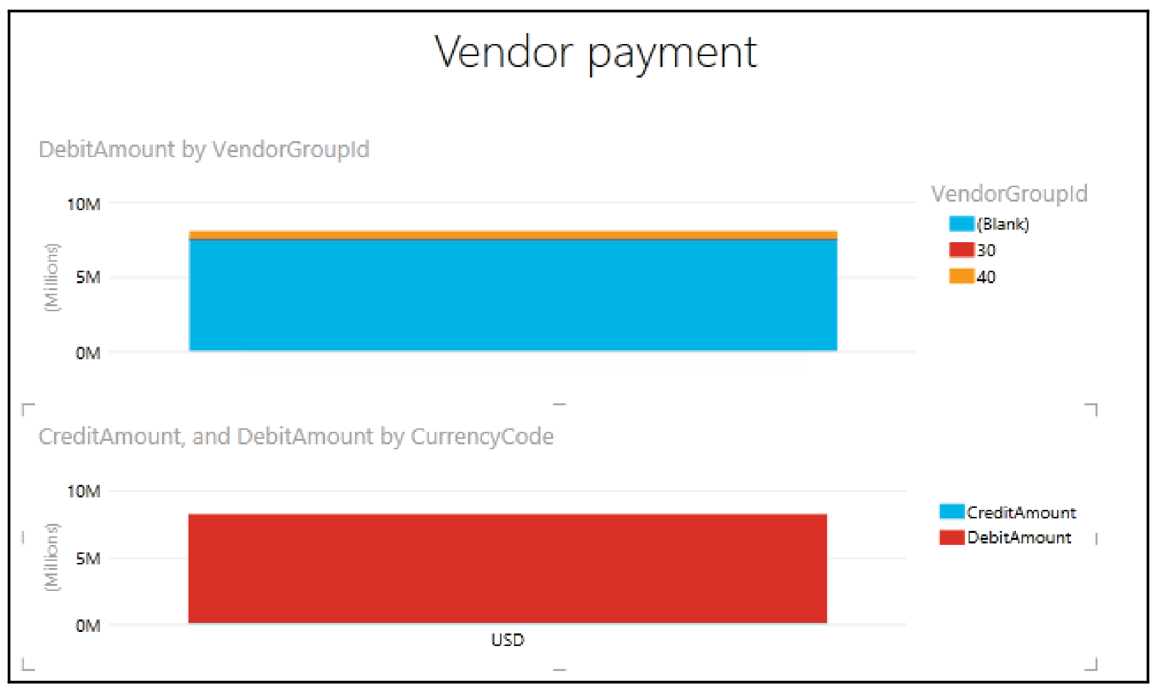
- Click on another cell below the previous one, as shown in the following screenshot, and enter the following formula:

```
Total Credit:=CALCULATE(SUM([CreditAmount]))
```

6. Now we need to add a report using this query, navigate to **Insert** | **Power View**. It will load a blank power view report. You will get your query in this report. Now drag and drop the total fields that we created in the previous step. Also, you can update the design using **Design** | **Switch visualization**:



7. In case you are looking for a specific view, you can try to drag the fields in the **Power View** pane in the left window, and drag them into the correct Axis and Legend areas to get the required visuals:



8. You can now drill into the vendor group by double-clicking the `VendorGroupId` column.

How it works...

Using OData end points, it's easy to expose into Excel. To validate that data entities are exposed by using the OData v4 endpoint, add a suffix of `data` to the URL. The URL should now be in the following format:

```
https://yourenvironment.cloudax.dynamics.com/data
```

For Internet Explorer, JSON files need to be downloaded before you can view them. Use Visual Studio to view the JSON file. If you're using Chrome, you can view the feed in your browser window itself:

```
data.json
Schema: <No Schema Selected>
{
  "@odata.context":"https://dpkdevaxaos.cloudax.dynamics.com/data/$metadata","value":[
    {
      "name":"MCRSearchParameters","kind":"EntitySet","url":"MCRSearchParameters"
    },{
      "name":"ExciseTaxRegistrationNumbers","kind":"EntitySet","url":"ExciseTaxRegistrationNumbers"
    },{
      "name":"SecurityUserRoleOrganizations","kind":"EntitySet","url":"SecurityUserRoleOrganizations"
    },{
      "name":"RetailReturnPolicies","kind":"EntitySet","url":"RetailReturnPolicies"
    },{
      "name":"DeliveryModeChannelLines","kind":"EntitySet","url":"DeliveryModeChannelLines"
    },{
      "name":"BankStatementLines","kind":"EntitySet","url":"BankStatementLines"
    },{
      "name":"JobADARequirements","kind":"EntitySet","url":"JobADARequirements"
    },{
      "name":"DataManagementTemplateDetails","kind":"EntitySet","url":"DataManagementTemplateDetails"
    },{
      "name":"CallCenterCatalogCoupons","kind":"EntitySet","url":"CallCenterCatalogCoupons"
    },{
      "name":"IntrastatSpecialMovements","kind":"EntitySet","url":"IntrastatSpecialMovements"
    },{
```

It will show all available queries that are available.

There are a few more add-ins on the market such as *Microsoft SQL Server Power Pivot for Excel*, which enables data modeling for business users using an in-memory engine that is built into Excel. This will allow you to process large amounts of data at the same time without worrying about performance.

Another add-in is *Power View*, which lets you build visualizations that are interactive in Excel itself by using data modeled and loaded by Power Pivot and Power Query.

See also

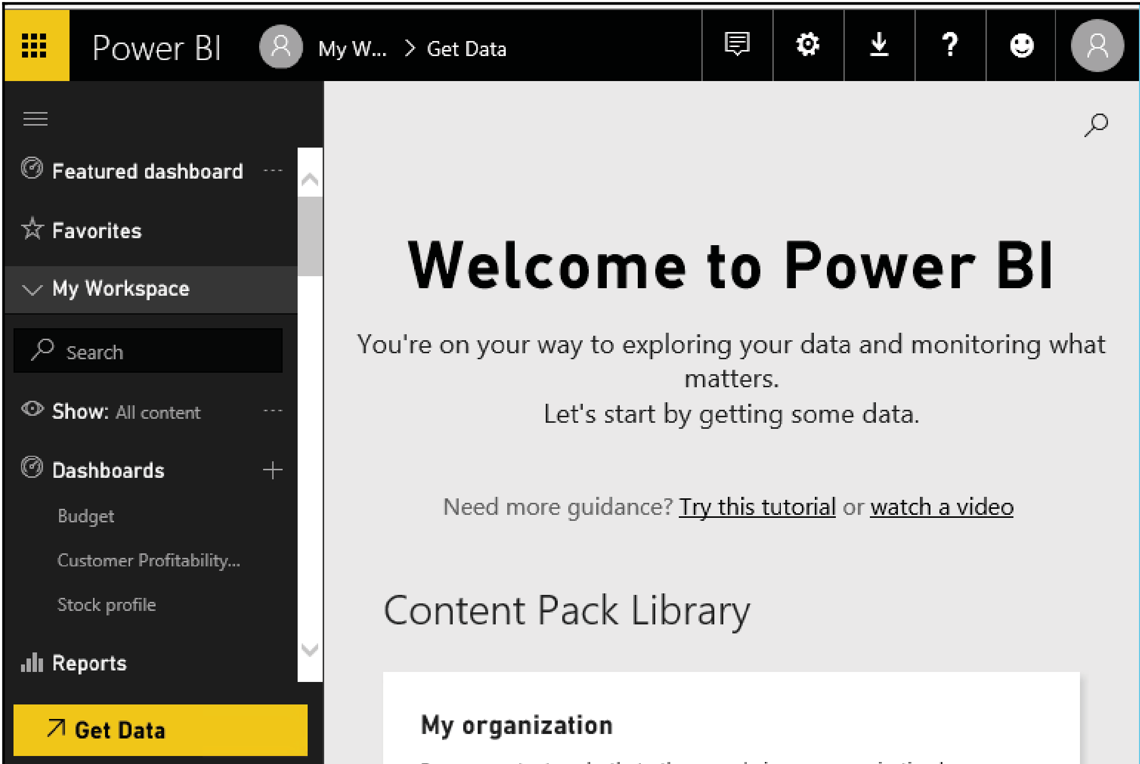
- If you are not getting a Power view option in Excel, follow this link to enable it: `https://support.office.com/en-us/article/Turn-on-Power-View-in-Excel-2016-for-Windows-f8fc21a6-08fc-407a-8a91-643fa848729a`

Integrating Excel with Power BI

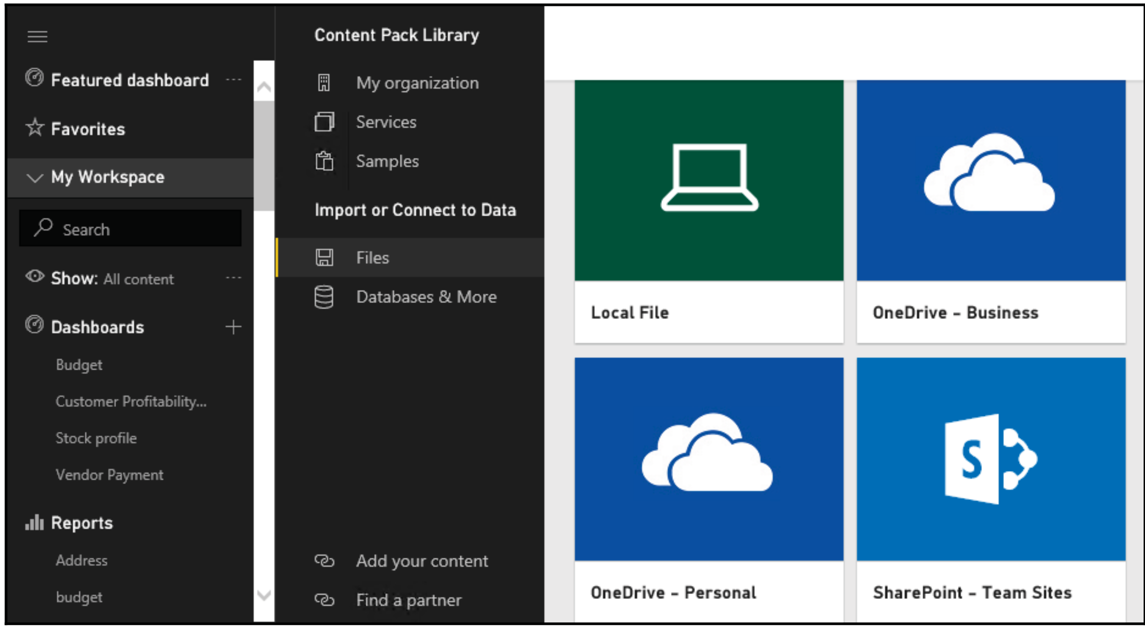
In this recipe, we will continue from the last recipe and create a dashboard by using this data model. Use this excel file as a data source to build your Power BI tile.

How to do it...

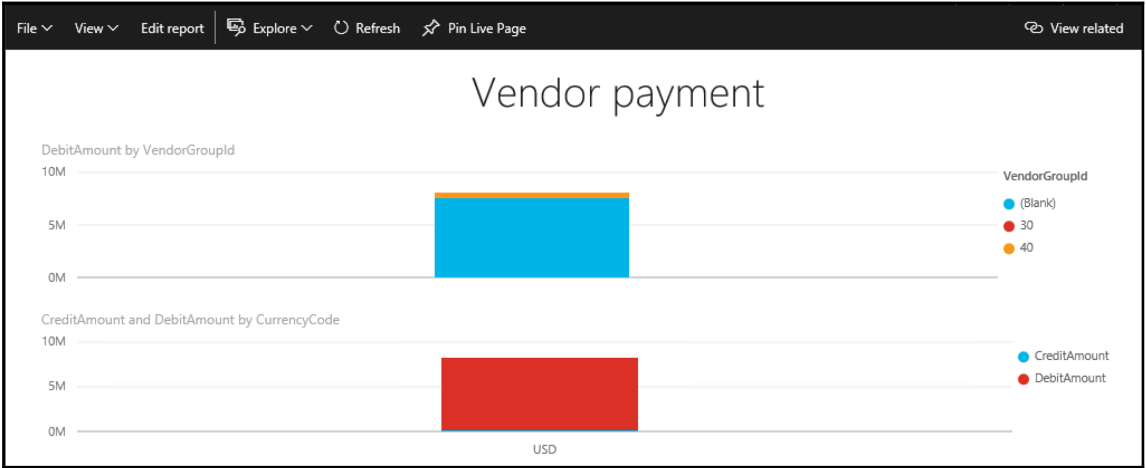
1. Log in to `http://app.powerbi.com` using your organization credentials. For first time users, you can sign up for a trial version:



2. Create a new dashboard by clicking on the plus sign (+) next to **Dashboards**. Enter the name as `vendor payment`.
3. Click on **GetData** and under the **Files** tile click on select **Local File** and load your excel file and then import it into Power BI:

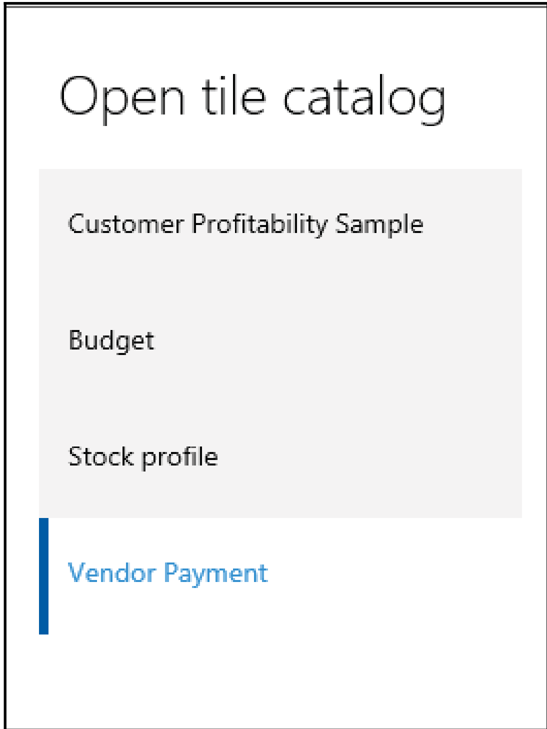


4. On successful import your dashboard should look like this:



Save your setup.

5. Now you will be able to use this Power BI tile in Dynamics 365 for Operation workspaces:



To get this tile live on your WorkSpace you need to use the Power BI section in its respective workspace.

How it works...

Dynamics 365 for operation data source is the most useful data source for Power BI as Power BI authoring tools, like Excel and Power BI Desktop. You have to connect to data using an OData feed that uses Power Query and access data entities that can be used to author Power BI reports and dashboards.

OData V4 authentication is required to expose Dynamics 365 for Finance and Operations data entities as data feeds. Data entities included here are standard data entities and aggregate data entities that enable you to summarize and calculate data in the OData feed. All business users can get access to these feeds based on the security privileges defined in the system. OData authentication can also be consumed by any other tool that supports the OData protocol and works with Dynamics 365 for Finance and Operations data securely.

See also

- If you are using Microsoft Excel 2013, you must download Power Query using the following link:
  `http://www.microsoft.com/en-gb/download/details.aspx?id=39379`

Developing interactive dashboards

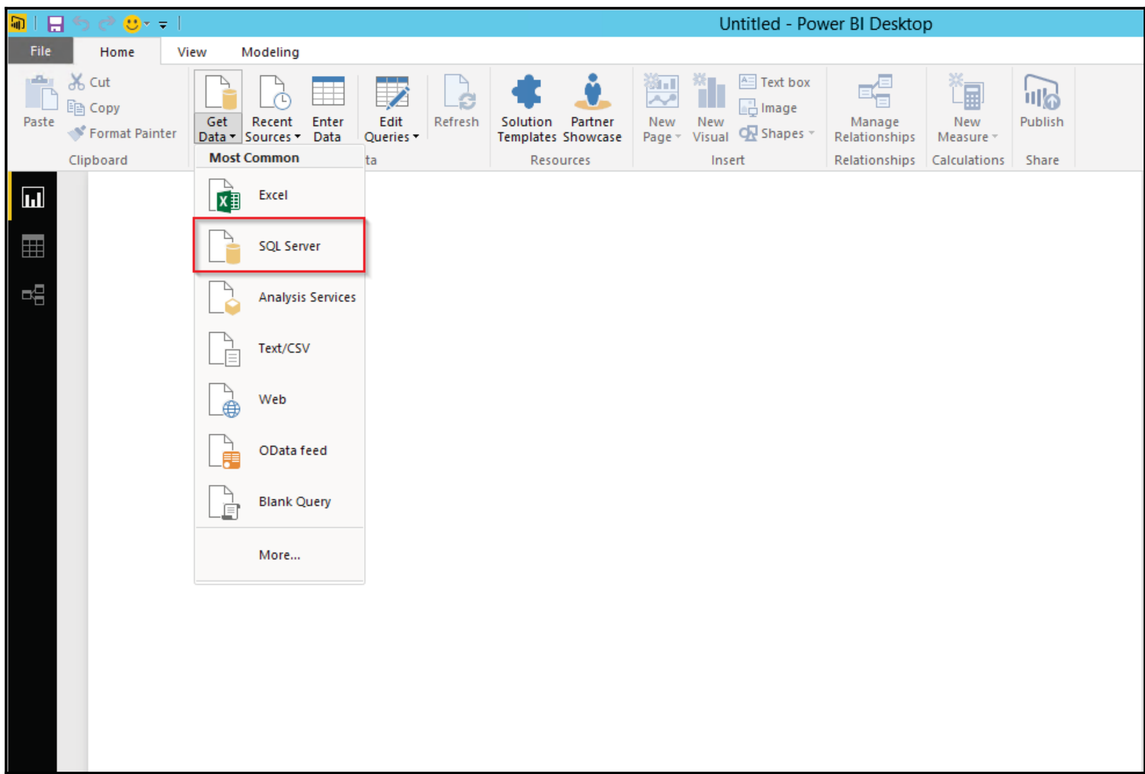
Microsoft ships pre-built Dynamics 365 for Operation content packs for Power BI, which includes dashboards and reports. For demonstration purposes we could also load these content packs from `PowerBI.com` in our application and analyze the data.

In this recipe, we will show you how to develop a Power BI dashboard for organizational content packs, to share their data and datasets, reports, and other visual analytical data within their organization.

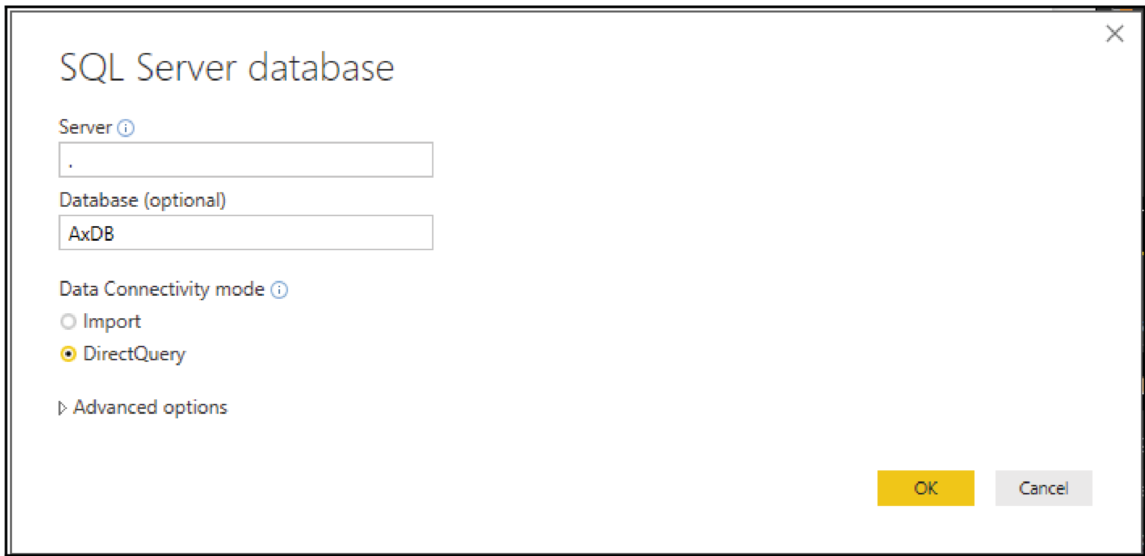
How to do it...

1. Get Power BI desktop to connect your Azure SQL database from this link: `https://www.microsoft.com/en-us/download/details.aspx?id=45331`.

2. After installation, open Power BI Desktop and select the **Get Data** option, as shown in the following screenshot:

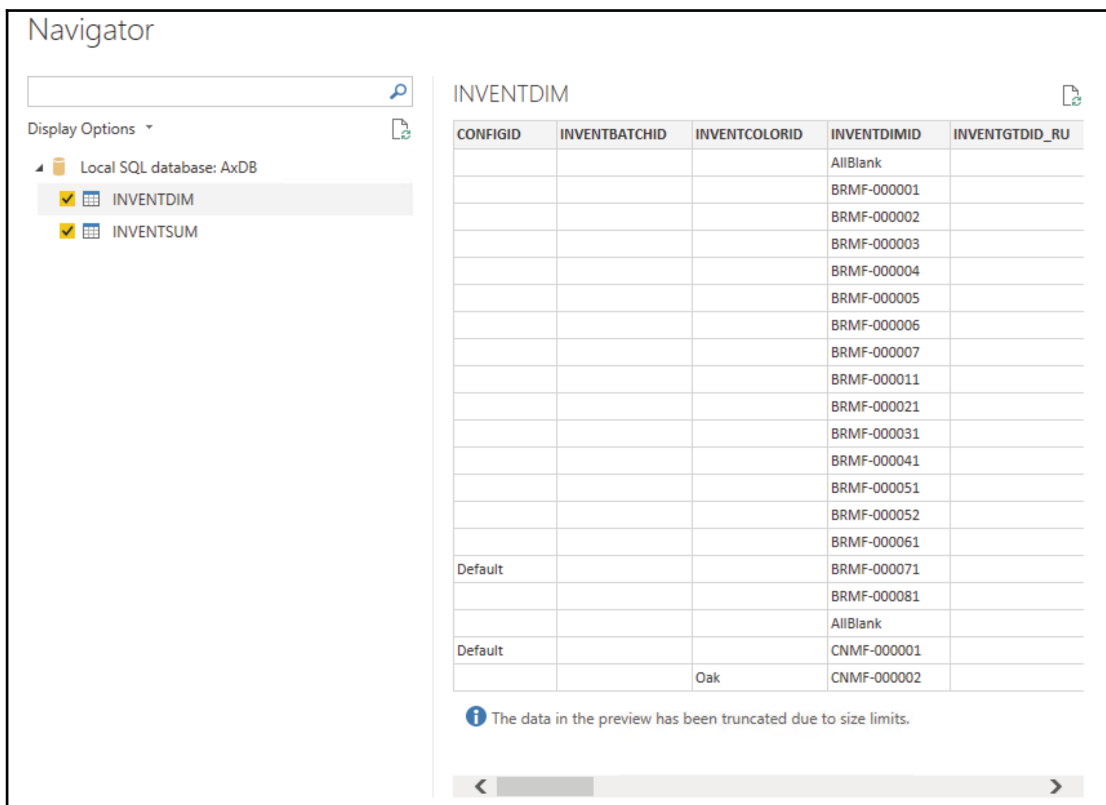


3. Here we get various options for our data source. We could choose any of the options here, but for this recipe we will choose SQL Server:

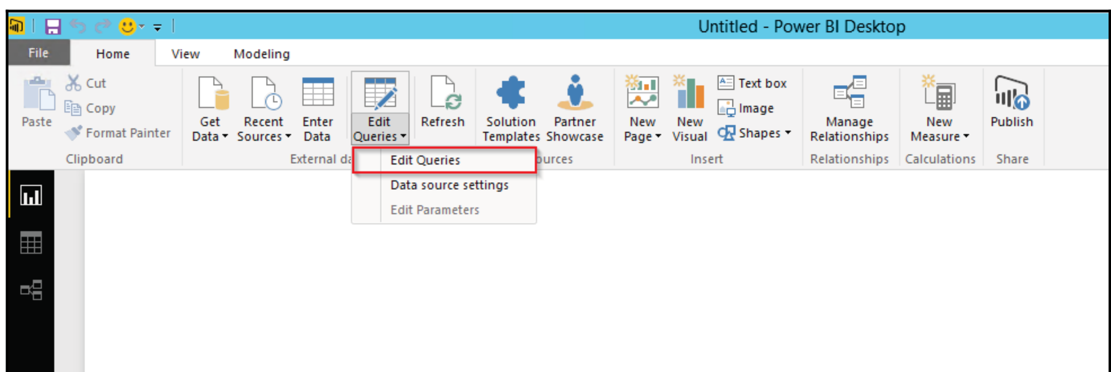


4. Fill in the details as shown in the preceding screenshot and click **OK** to connect.

5. Select the two tables, **InventSum** and **InventDim**, as shown in the screenshot and click **Load**:

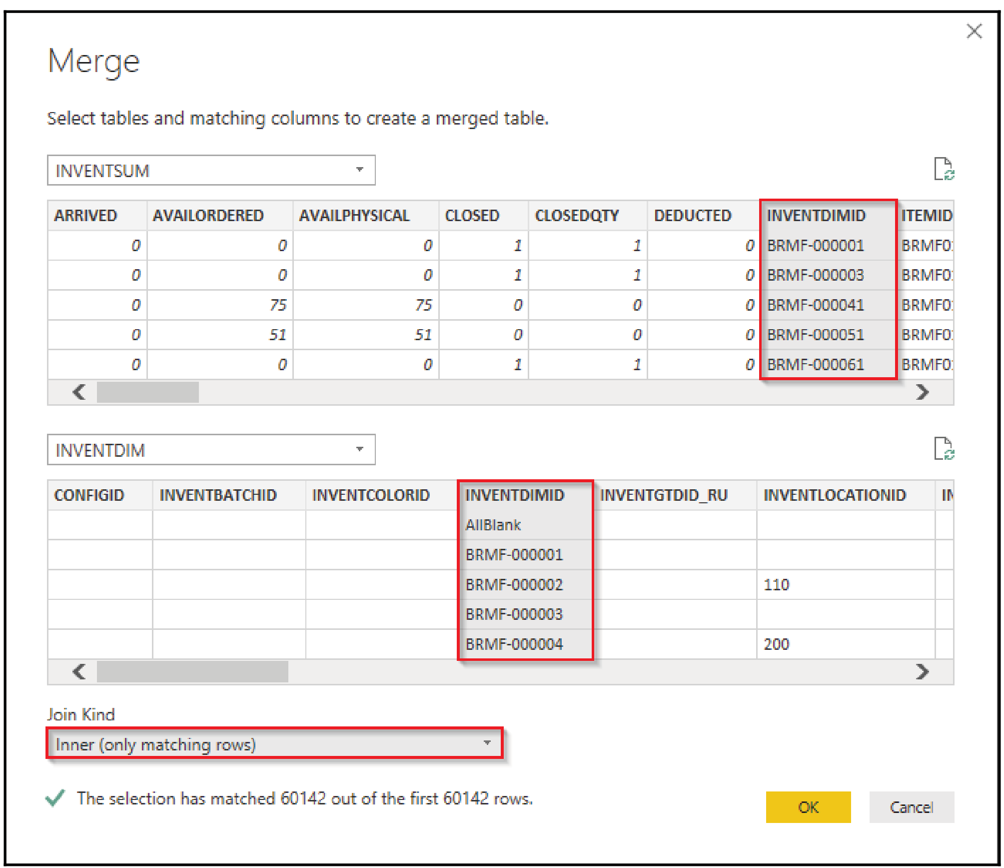


6. Select the **Edit Queries** option, as shown in the following screenshot:

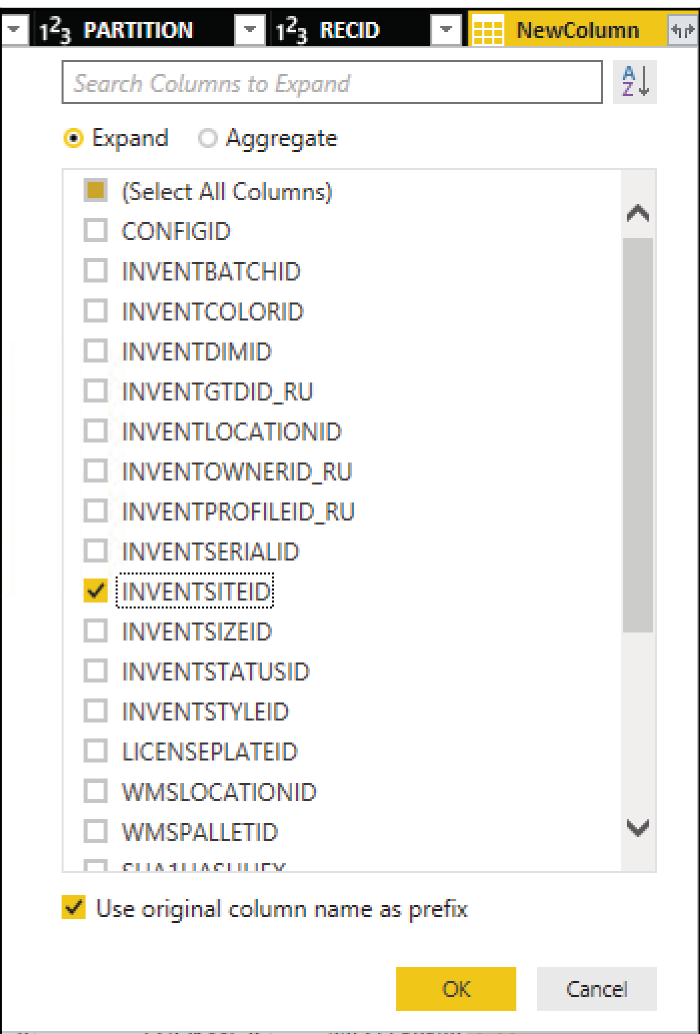


It will open up Query editor, select **Home** | **Combine group** | **Merge queries** under the combine tab, as shown in the following screenshot.

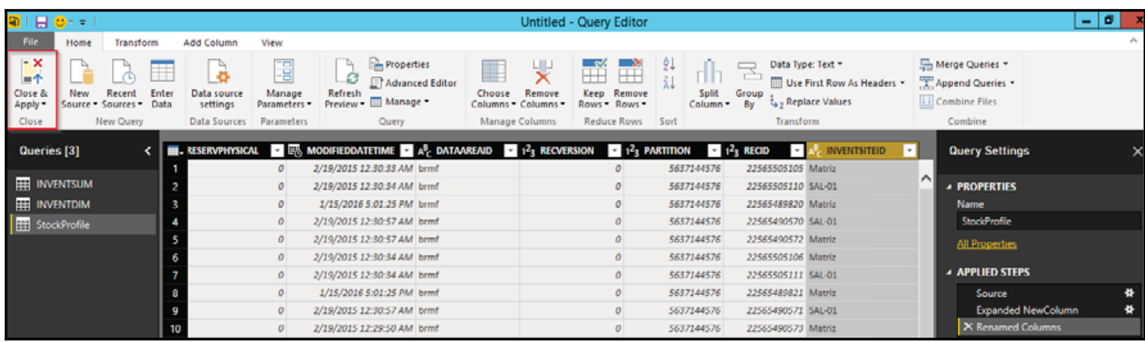
7. Under the **Merge** queries window, select the values as shown in the following screenshot:



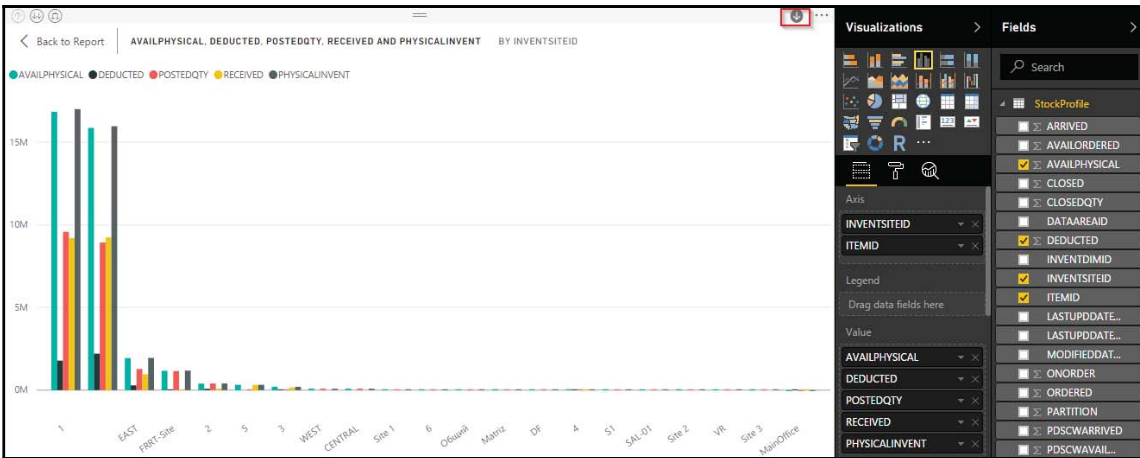
8. Here, we need to specify relationships between the two data sources on the **InventDimId** field and select **Join** type as **Inner join**. On the new merged query **Stock profile**, select the specific column field as **INVENTSITEID**:



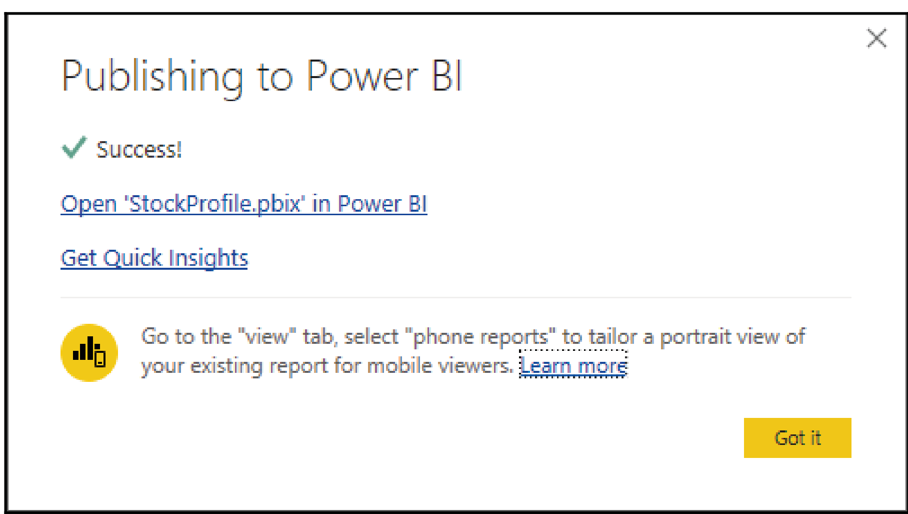
9. On the **Query Editor** window, select **Close & Apply**:



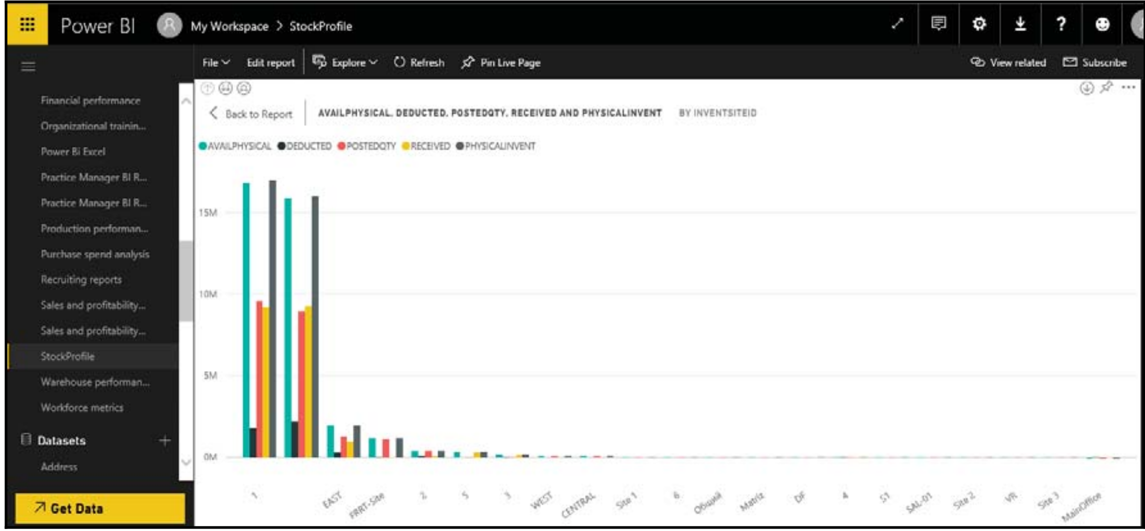
10. Select Aggregate fields such as **AVAILPHYSICAL**, **DEDUCTED**, **POSTEDQTY**, **RECEIVED,** and **PHYSICALINVENT**. Select fields **INVENTSITEID** and **ITEMID** in AXIS and turn on drill down mode as shown in the following screenshot:



11. Publish the report to Power BI using the **Publish** button on the **Share** tab. Use a power BI account to publish the report:



12. Log in to `powerbi.com` and find the report **StockProfile** under the **Reports** tab:



How it works...

Here in this recipe, we load the data of **InventSum** and **InventDim** by connecting to the Dynamics 365 for Finance and Operations SQL database. Once the queries for the two tables are loaded then we merge the queries into one by specifying **Inner Join** on the **InventDim** field.

Here we fetch the **InventSiteId** field from the **InventDim** table and provide the latter on the AXIS of the chart. In addition to this we provide an **ItemId** field on the AXIS as well under **InventSiteId** for drill down on stock of items on site.

Embedding Power BI visuals

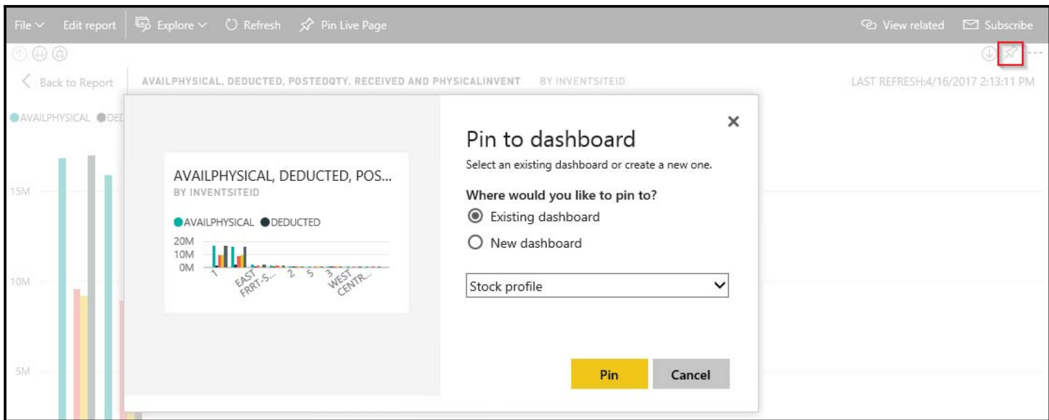
Dashboard solutions and reporting can be crafted and viewed on their own in Power BI for any organization and business unit. Dynamics 365 for operation provides Power BI columns on some of the workspaces. Just click **Get Started** and it opens a Power BI tile catalog; these are the dashboard solutions and reports that are pinned on Power BI.

In this recipe, we will show you how you can pin interactive dashboards developed on Power BI to a Dynamics 365 for Finance and Operations workspace to provide intuitive visuals. We could also open `PowerBI.com` from here by clicking on dashboard for a more interactive analysis.

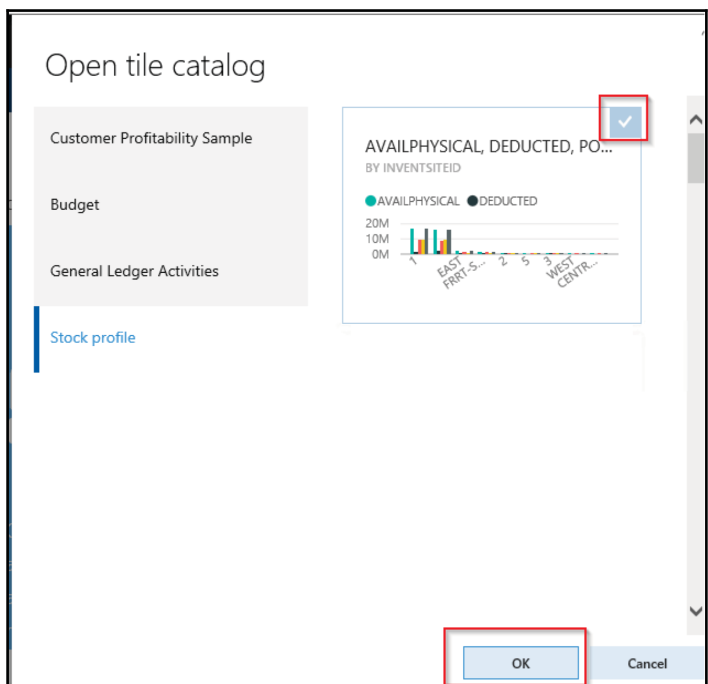
How to do it...

1. Log in to `PowerBI.com` and create a new dashboard by using a plus (**+**) sign to add a blank dashboard.

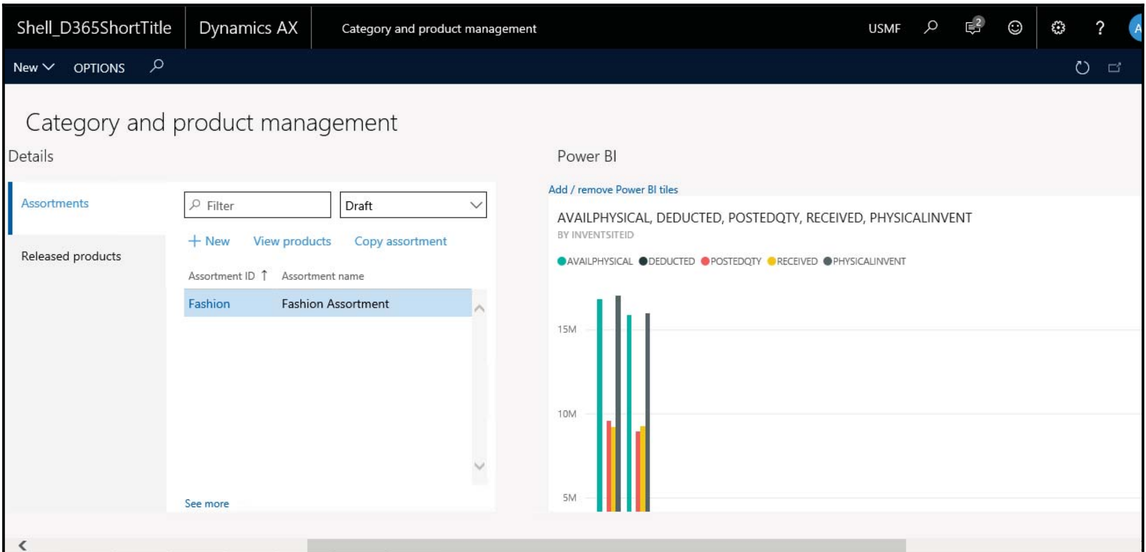
2. Access our report **Stock profile** and select the option **Pin** visual on the report:



3. Now log in to Dynamics 365 for Finance and Operations and move to **Category and product management** workspace and connect to Power BI.
4. Select the tile for **Stock profile** and press **OK** to embed it in Dynamics 365 for Finance and Operations:



5. This embeds the screen on our workspcae. Furthermore, for analysis and drill-down purposes we could double-click on the report and open it in `PowerBI.com` for more interactive analysis:



How it works...

In the preceding recipe, we have obtained analytical data by querying an SQL database. The query is used to aggregate and calculate data on **InventSum** and create a power BI tile. We are already signed into Dynamics 365 for Finance and Operations using **AAD** (**Azure Active Directory**), so authorization of a connection between Dynamics 365 for Finance and Operations and Power BI is done already. Power BI tiles and visualizations are loaded, which we could pin to the workspace of Dynamics 365 for Finance and Operations.

9

Integration with Services

In this chapter, we will cover the following recipes:

- Authenticating a native client app
- Creating a custom service
- Consuming custom services in JSON
- Consuming custom services in SOAP
- Consuming OData services
- Consuming external web services

Introduction

In a cross-platform environment and large enterprises, there are many scenarios wherein many external or third-party solutions are used, which might be a web application or an application inside a domain. They need to integrate data with Dynamics 365 for Finance and Operations in real time or at specified intervals depending on the nature of the business, such as currency exchange rates from banks, syncing with POS terminals, and inbound/outbound integrations with other legacy systems.

Dynamics 365 provides us with ample options that we can use for integrating, such as custom services that we create to expose X++ business logic through a service interface for inbound and outbound integrations, OData REST endpoints that encapsulates Dynamics 365 for Finance and Operations business entities and allows us to perform CRUD operations using OAuth V2.0.

In this chapter, we will learn to use various integration technologies using simple business solution examples that are required day-to-day.

Authenticating a native client app

Azure Active Directory (**AAD**) uses OAuth 2.0 to enable you to authorize access to web applications and web APIs in your Azure AD tenant. This guide is language independent, and describes how to send and receive HTTP messages without using any of our open-source libraries.

OData Services, JSON-based custom services, and REST metadata services support standard OAuth 2.0 authentication.

Getting ready

Although we can create multiple types of apps using ADD, here we will discuss two kinds of applications that are supported in Microsoft AAD for Dynamics 365 for Operation:

- **Native client application**: This requires a redirect URI, which Azure AD uses to return token responses. This flow uses a username and password for authentication and authorization.
- **Web App/API (Confidential client)**: A confidential client is an application that can keep a client password confidential to the world. It uses a client app ID and a client secret key to prepare client credentials. The authorization server assigns this client password to the client application.

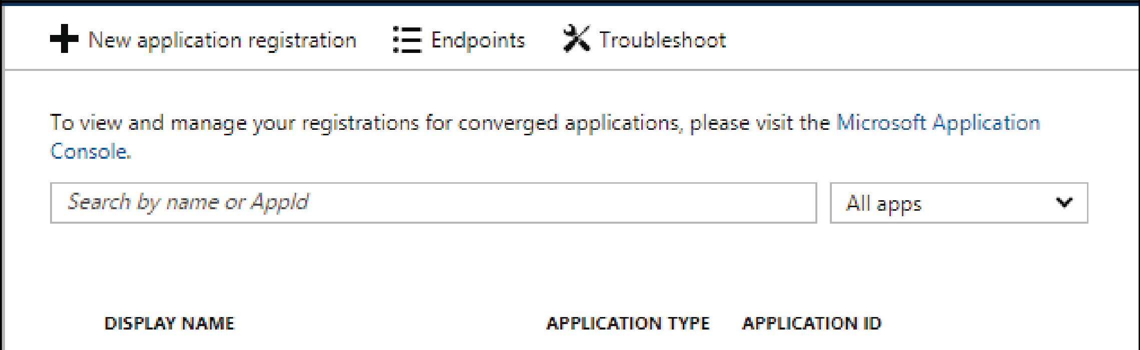
These details are shown on Microsoft's official website.

Registering an app with ADD will create an application ID for a new application and also enables it to receive tokens from external systems. Let's see how we can add a new native app in Azure Active Directory. In this recipe, get the help of the NuGet Package Manager Console to install the `Microsoft.IdentityModel.Clients.ActiveDirectory` library.

To get more details on how to install the `Microsoft.IdentityModel.Clients.ActiveDirectory` library, visit `https://www.nuget.org/packages/Microsoft.IdentityModel.Clients.ActiveDirectory`.

How to do it...

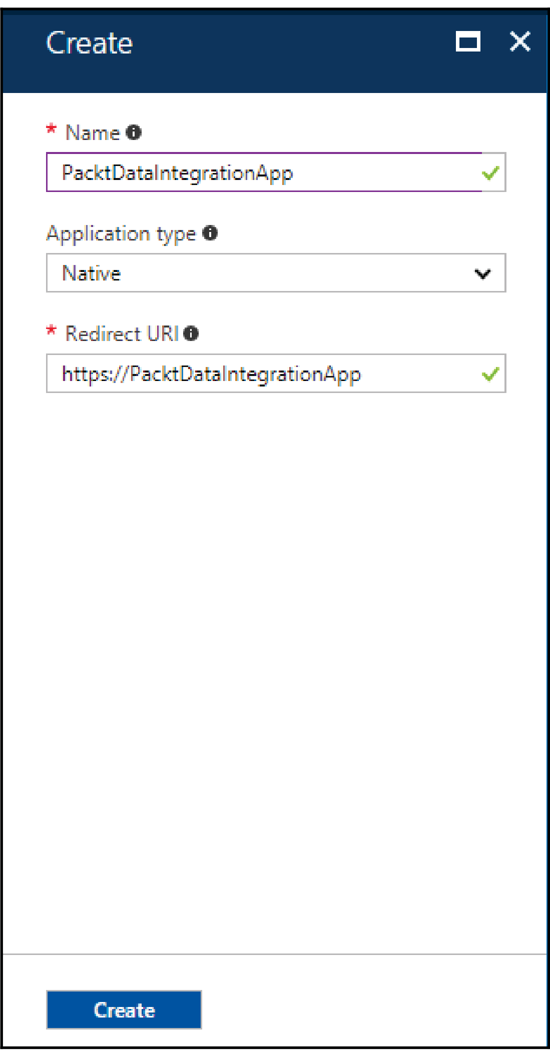
1. Log in to the Azure portal (`https://portal.azure.com`).
2. If you have multiple AAD tenants, select the one that you want to use to create a new app.
3. Go to Menu in the left corner of the portal and select **Azure active directory** | **App registration**. The following screenshot shows the form to add a new application:



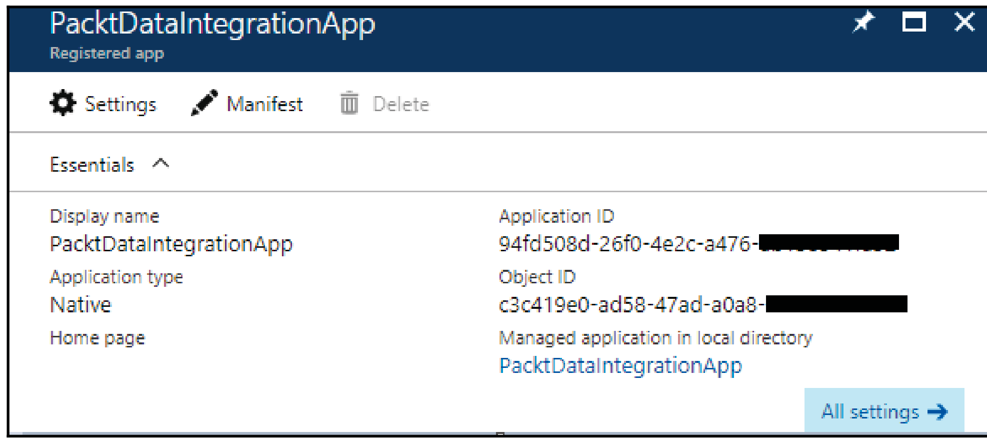
4. Click on **New application registration**, and, in the new form, fill in all the details as in the following table and hit the **create** button:

Name	**PacktDataIntegarionApp**
Application type	Native
Redirect URI	`https://PacktDataIntegrationApp`

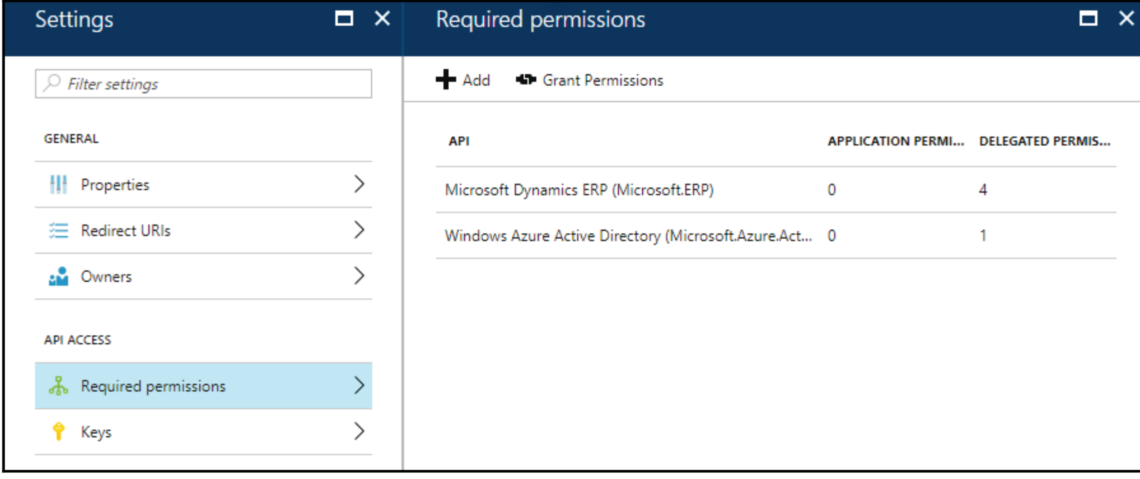
Your form must look as follows:



5. On completion of this registration, AD assigns your application a unique client identifier, that is, **Application ID** to your application:



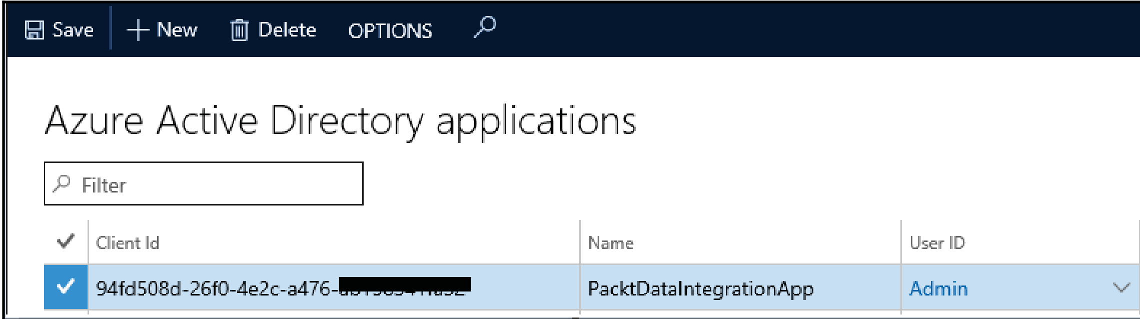
6. Check the settings and make sure all other properties for this new application are in place before using it. Refer to the following screenshot:



7. Now let's register this app in Dynamics 365 for Finance and Operations. Navigate to **System administration** | **Setup** | **Azure Active Directory applications** form, click on the new button and fill in the required details as follows:

Client Id	`94fd508d-26f0-4e2c-a476-********`
Name	`PacktDataIntegrationApp`
User Id	`Admin`

8. Your record should look as follows:



After this final step, we are now good to use this app in our code.

9. Now, create a new Visual Studio C# class library project, which we will utilize for getting authentication to access Dynamics 365 for Finance and Operationss ERP. Later, we will utilize the same to authenticate access to Dynamics 365 for Finance and Operations by our web service.
10. Create a new class `ClientConfiguration.cs` ,where we will specify our configuration using properties. Specify the following namespaces:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

11. Add the following specified code in our class. Here, we will set the default values of properties:

```
public partial class ClientConfiguration
{
  public static ClientConfiguration Default {get {return
```

```
     ClientConfiguration.OneBox;}}
    public static ClientConfiguration OneBox = new
     ClientConfiguration()
    {
      UriString =
       "https://d365devdpkcdfe0b0****1caos.cloudax
        .dynamics.com/",
      ActiveDirectoryResource =
       "https://d365devdpkcdfe0****01caos.cloudax.dynamics.com",
      ActiveDirectoryTenant =
       "https://login.windows.net/myTenant.onmicrosoft.com",
      ActiveDirectoryClientAppId = "81dada10-f7ee-4fe3-a6b2-
       5******",
    };
    public string UriString { get; set; }
    public string ActiveDirectoryResource { get; set; }
    public String ActiveDirectoryTenant { get; set; }
    public String ActiveDirectoryClientAppId { get; set; }
  }
```

12. We will create a new class, where we will perform actual operations to get authentication, and name this class as `OAuthHelper`:

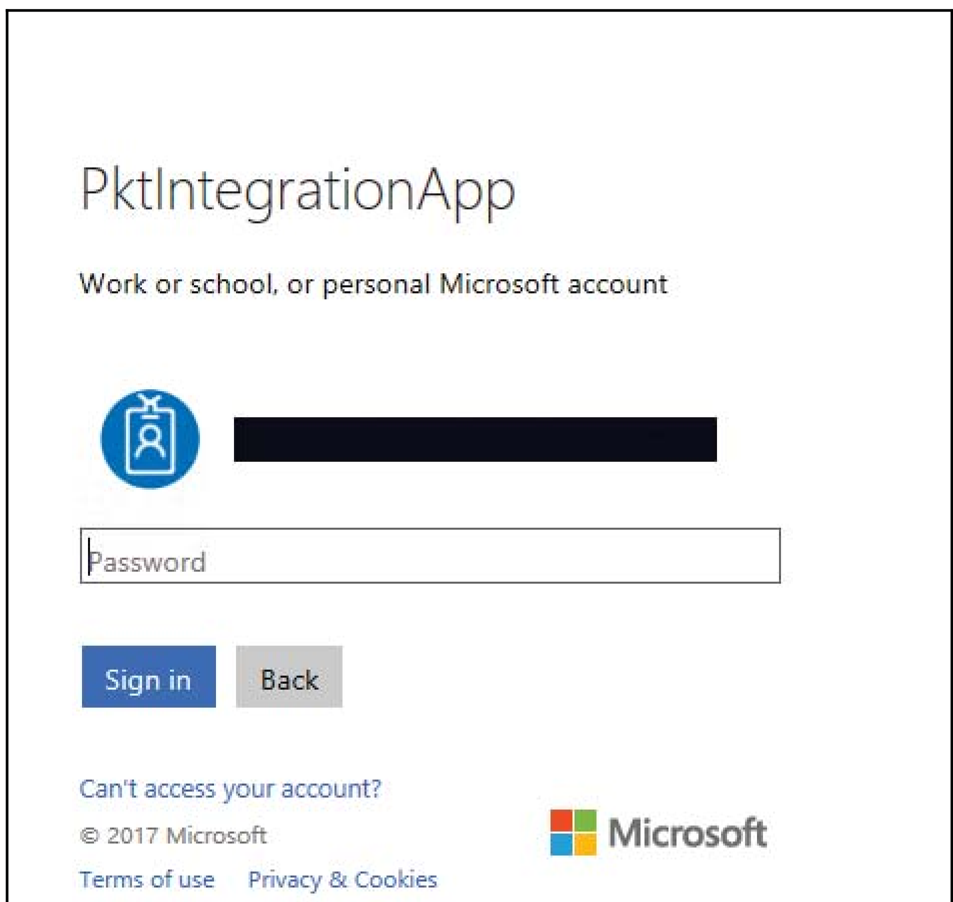
```
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

public class OAuthHelper
{
  /// <summary>
  /// The header to use for OAuth.
  /// </summary>
  public const string OAuthHeader = "Authorization";
  /// <summary>
  /// retrieves an authentication header from the service.
  /// </summary>
  /// <returns>the authentication header for the Web API call.
      </returns>
  public static string GetAuthenticationHeader(bool
   useWebAppAuthentication = false)
  {
    string aadTenant =
      ClientConfiguration.Default.ActiveDirectoryTenant;
    string aadClientAppId =
      ClientConfiguration.Default.ActiveDirectoryClientAppId;
```

```
        string aadResource =
          ClientConfiguration.Default.ActiveDirectoryResource;
        AuthenticationResult authenticationResult;
        var authenticationContext = new
         AuthenticationContext(aadTenant,
          TokenCache.DefaultShared);
        authenticationResult =
          authenticationContext.AcquireTokenAsync(
            aadResource, aadClientAppId,
            new Uri("https://packtIntegrationApp"),
            new PlatformParameters(PromptBehavior.Auto)).Result;
        // Create and get JWT token
        return authenticationResult.CreateAuthorizationHeader();
      }
    }
```

13. Now, to test the authentication, we could create a new console application, name it `PacktTestAuthentication` ,and set it as a start-up project. Add new code in our `Program.cs` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using AuthenticationUtility;

namespace PacktTestAuthentication
{
  class Program
  {
    static void Main(string[] args)
    {
      var oauthHeader = OAuthHelper.GetAuthenticationHeader();
      Console.WriteLine(oauthHeader.ToString());
      Console.ReadLine();
    }
  }
}
```

14. Now, click on the **Start** button in VS and you will be prompted to enter the username and password of the Dynamics 365 for Finance and Operations user:



15. It will present you with an Authentication header, as shown in the following screenshot:

You need this authentication header while interacting with this app.

How it works...

Azure active directory provides a secure sign in and authorization for its services. Dynamics 365 for Finance and Operations is also based on a **SaaS** (**Software as a Service**) module, and Azure plays an important role in providing all required prerequires for your environment.

When you register a new app in AAD, it creates an application ID for it and this can be used to register in Dynamics 365 and in your code to enable a bridge between Dynamics 365 and your application. While calling this application in a console app, you get an interface where you can use your system credentials to get an authentication header.

Here, while registering the app, we are presented with options to select **Web Application** or **Native Client Application**. We will be utilizing this app for authentication in console applications, so here we are using **Native Client Application**.

We create a class `ClientConfiguration.cs` to specify the configuration for accessing authorization header. We create properties for URI, active directory resource, aadTenant, and azure active directory client app ID and specify their default values. Next, we create an authorization helper class `OAuthHelper.cs` . Here, we first create an object of authorization context by calling `AuthenticationContext(String, TokenCache)` that will return us the context with the address of the authority. Passing `TokenCache.DefaultShared` means that the authentication library will automatically save tokens in default `TokenCache` whenever we obtain them.

There's more...

If you are using web applications, provide the sign-on URL, which must be the base URL of your app, where users can sign in, for example, `http://localhost:8888`.

To acquire a security token from the authority, we use the overloaded method `AcquireTokenAsync` , where we pass `aadResource`, client app ID, and the redirect URI specified on the native client app:

- **aadResource**: Identifies the target resource that is the recipient of the requested token
- **client App Id**: Identifies the Azure client app that has the requested token
- **redirect URI**: Identifies the URI address that needs to be returned on response from the Azure authentication authority
- **PlatformParameters**: Identifies object of additional parameters used for authorization

Next, we get an object of the `authorizationResult` class and call its method `CreateAuthorizationHeader`, which returns an authorization header.

See also

- App sample on GitHub: `https://github.com/Azure-Samples?utf8=%E2%9C%93&query=active-directory`
- Code sample: `https://docs.microsoft.com/en-gb/azure/active-directory/develop/active-directory-code-samples`

Creating a custom service

In Dynamics 365 for Finance and Operations, we still have custom services available to expose the system's functionality to the external world. Microsoft uses the SysOperation framework, where data contracts are decorated with standard attributes, and it automatically serializes and desterilizes data that is shared between two applications. Any service method can be exposed to the external world by using `SysEntryPointAttribute` and then specifying the service operation under **Application Explorer** | **AOT** | **Services**.

In this recipe, we will create a custom service that will be consumed in subsequent recipes to demonstrate how it could be consumed by the external world.

Getting ready

For this recipe, you should have a little knowledge of data contracts and service methods.

How to do it...

1. Create a new Dynamics 365 for Finance and Operations project in Visual Studio and name it `BuildingCustomService`.
2. In the project, create a new class, which will serve as a data contract for our custom web service. Name this class `PacktCustBalanceDataContract`. This class is decorated with the `DataContract` attribute and the data methods are decorated with `DataMemberAttribute`. We will create the following data methods in our class, which will be exposed to the external world as parameters:

```
[
  DataContractAttribute, SysOperationGroupAttribute('Date',
   "@ApplicationPlatform:SingleSpace", '1')
]
class PacktCustBalanceDataContract
{
  TransDate   transDate;
  CustAccount accountNum;
  DataAreaId dataAreaId;

  /// <summary>
  /// Gets or sets the value of the datacontract parameter
      DateTransactionDate.
  /// </summary>
  /// <param name="_transDate">
```

```
/// The new value of the datacontract parameter
    DateTransactionDate;
/// </param>
/// <returns>
///  The current value of datacontract parameter
     DateTransactionDate
/// </returns>
[DataMemberAttribute('DateTransactionDate'),
 SysOperationLabelAttribute(literalStr("@SYS11284")),
 SysOperationGroupMemberAttribute('Date'),
 SysOperationDisplayOrderAttribute('1')] // today's date
public TransDate parmTransDate(TransDate _transDate =
  transDate)
{
  transDate = _transDate;

  return transDate;
}

[DataMemberAttribute('Company'),
  SysOperationLabelAttribute(literalStr("@SYS11284")),
  SysOperationGroupMemberAttribute('Company'),
  SysOperationDisplayOrderAttribute('3')] // today's date
public DataAreaId parmDataAreaId(DataAreaId _dataAreaId =
  dataAreaId)
{
  dataAreaId = _dataAreaId;

  return dataAreaId;
}

/// <summary>
/// Gets or sets the value of the datacontract parameter
    CustomerAccount.
/// </summary>
/// <param name="_accountNum">
/// The new value of the datacontract parameter
    CustomerAccount;
/// </param>
/// <returns>
///  The current value of datacontract parameter
     CustomerAccount
/// </returns>
[DataMemberAttribute('CustomerAccount'),
 SysOperationLabelAttribute(literalStr("Account number")),
 SysOperationGroupMemberAttribute('Account'),
 SysOperationDisplayOrderAttribute('2')]
public CustAccount parmCustAccount(CustAccount _accountNum =
```

```
        accountNum)
      {
        accountNum = _accountNum;

        return accountNum;
      }
    }
```

3. Now, let's create a service method that will accept a data contract as a parameter and returns back the balance of the customer:

```
class PacktCustBalanceService
{
  [AifCollectionType('return', Types::Real,
   extendedTypeStr(Amount))]
  public Amount processData(PacktCustBalanceDataContract
   _custBalanceDataContract)
  {
    QueryRun    queryRun;
    CustTable   custTable;
    Amount      balance;
    System.Exception ex;

    try
    {
      if(_custBalanceDataContract.parmDataAreaId())
      {
        changecompany(
          _custBalanceDataContract.parmDataAreaId())
        {
          // create a new queryrun object
             var query = new Query();
             var qbds =
               query.addDataSource(tableNum(CustTable));
             qbds.addRange(fieldNum(
               CustTable,AccountNum)).value(
                 _custBalanceDataContract.parmCustAccount());
             queryRun = new queryRun(query);
          // loop all results from the query
             while(queryRun.next())
             {
               custTable = queryRun.get(tableNum(custTable));
               // display the balance
               balance = custTable.balanceMST();
             }
        }
      }
    }
```

```
            catch (Exception::CLRError)
            {
              ex = ClrInterop::getLastException();
              if (ex != null)
              {
                ex = ex.get_InnerException();
                if (ex != null)
                {
                  error(ex.ToString());
                }
              }
            }
            return balance;
          }

        }
```

4. Create a new service object under the services node with the name `PacktCustBalanceService` and set the following properties on it:

Property	**Value**
Class	`PacktCustBalanceService`
External name	`PacktCustBalanceService`
Namespace	`http://schemas.packt.com/CustBalance`

5. Right-click on the **Service Operations** node under our service `PacktCustBalanceService` , select new service operations and set the following properties:

Method	`processData`
Name	`processData`

6. Create a new object under the **Service** groups node and name it `PacktCustBalanceServices`. Right-click and select **New Service** . Set the following properties on it:

Name	**Service**
Service	`PacktCustBalanceService`

7. Build the project and, on successful build, our web service is available to the external world.

How it works...

Once the project is built, our web service is deployed on SOAP and JSON endpoints automatically. Now, the contract class that we created here serves the purpose of passing the parameters that are required by our service for processing business logic. The data member attribute on the method in the contract class tells the system this method needs to be exposed to the external world as a parameter.

Our service method `processData` has the return type Amount EDT, but the external world doesn't know about this type, so we have used the `AifCollectionType` attribute to return the type Real.

Consuming custom services in JSON

Custom services are always deployed on two endpoints:

- **JSON** (**JavaScript Object Notation**) endpoint
- **SOAP** (**Simple Object Access Protocol**) endpoint

In Dynamics 365 for Finance and Operations, all service groups under **AOT** | **Service groups** are automatically deployed. Therefore, all services that need to be deployed must be a part of a service group.

JSON endpoint is deployed at `https://<host_uri>/api/Services/<service_group_name>/<service_group_service_name>/<operation_name>`.

In this recipe, we will consume Dynamics 365 for Finance and Operations custom web service in JSON.

Getting ready

You need to install Newtonsoft.Json NuGet to build this recipe. So, navigate to **NuGet Package Manager Console** in Visual Studio and install Newtonsoft.Json. As we have done in the previous recipe, get the help of NuGet Package Manager Console to install the Microsoft.IdentityModel.Clients.ActiveDirectory library.

To get more details on how to install the Microsoft.IdentityModel.Clients.ActiveDirectory library, visit https://www.nuget.org/packages/Newtonsoft.Json/.

How to do it...

1. Create a new **C# console application** in our project and name it PacktJsonApplication. Here, we need to add a reference to Newtonsoft.Json and our authentication project built in the first recipe.
2. Create a new class for a data contract, where we need to set the properties:

```
public class CustBalanceDataContract
{
  public string Company { get; set; }
  public string CustomerAccount { get; set; }
  public string DateTransactionDate { get; set; }
}
```

3. Create a new class RootObject for our datacontract that we need to serialize in JSON and pass as a parameter when calling the service:

```
public class RootObject
{
  public object CallContext { get; set; }
  public CustBalanceDataContract _custBalanceDataContract {
   get; set; }
}
```

4. Now, in our class `Program.cs`, which by default is added when we create the console application project, add the following namespaces:

```
using AuthenticationUtility;
using Newtonsoft.Json;
using System;
using System.IO;
using System.Net;
using System.Text;
```

5. Add the following code in our class `Program.cs`, which we will use to call the web service in JSON:

```
class Program
{
  public static string GetCustBalanceOperationPath=
    ClientConfiguration.Default.UriString +
     "api/services/PacktCustBalanceServices
     /Service/processData";

  static void Main(string[] args)
  {
    var custBalance = new CustBalanceDataContract();
    custBalance.CustomerAccount = "DE-001";
    custBalance.Company = "USMF";
    custBalance.DateTransactionDate = "0001-01-01T00:00:00";
    var rootJson = new RootObject();
    rootJson._custBalanceDataContract = custBalance;
    string json = JsonConvert.SerializeObject(rootJson);
    var request =
      HttpWebRequest.Create(GetCustBalanceOperationPath);
    request.Headers[OAuthHelper.OAuthHeader] =
      OAuthHelper.GetAuthenticationHeader();
    request.Method = "POST";
    request.ContentLength = 0;

    System.Text.UTF8Encoding encoding = new
      System.Text.UTF8Encoding();
    Byte[] byteArray = encoding.GetBytes(json);

    request.ContentLength = byteArray.Length;
    request.ContentType = @"application/json";

    using (Stream dataStream = request.GetRequestStream())
    {
      dataStream.Write(byteArray, 0, byteArray.Length);
    }
```

```
    long length = 0;
    try
    {
      using (HttpWebResponse response =
        (HttpWebResponse)request.GetResponse())
      {
        using (Stream responseStream =
          response.GetResponseStream())
        {
          using (StreamReader streamReader = new
            StreamReader(responseStream))
          {
             string responseString =
               streamReader.ReadToEnd();

             Console.WriteLine(responseString);
          }
        }
        length = response.ContentLength;
      }
    }
    catch (WebException ex)
    {
      //Write code to log the exception here.
    }

    Console.ReadLine();
  }
}
```

6. Let's run our application and check the results. Set the project as a startup project and click the **start** button on Visual Studio. You will get a prompt to add a username and password for authentication; after you sign in, you will get the result shown in the following screenshot:

```
Customer DE-001 balance is:382761.5
```

7. So, here, we get the balance of customer `DE-001` from USMF company.

How it works...

Here, we use `NewtonSoft.Json` to use the `JsonConvert` class to serialize C# object in JSON. The more difficult part is to prepare a JSON string that can be understood by D365 to fetch parameters. We first start by creating an object for `CustBalanceDataContract` and set contract parameters such as `CustomerAccount`, `Company`, and so on. Finally, we have to create a root contract that we will serialize and pass a JSON string to our web service. This contract consists of `dataContract` and it is called context contract. Here, the `HttpWebRequest.Create` method creates an instance of the JSON endpoint URI that we provide, and then we need to get authentication from azure to access our Dynamics 365 for Finance and Operations ERP application.

We specify `Method=POST` on our request to submit data to be processed by the JSON endpoint. MIME type for JSON is "**application/JSON**" and default encoding is **UTF8Encoding**, which we used to prepare the content and make objects of a request stream to write encrypted data on the request stream. In the next step, invoke the `GetResponse`, which gives us a `Webresponse` object, which we implicitly convert in the `HttpWebResponse` object, and then use it to get a response stream.

This is how we call the web service using JSON. You may note that calling a web service in JSON has less overheads and provides a significantly faster response than SOAP.

There's more...

Microsoft has introduced a REST (Representational State Transfer) metadata service in Dynamics 365 for Finance and Operations, which is a read-only service. In terms of OData, users can make only `GET` requests, since it is an OData implementation. The main purpose of this endpoint is to provide metadata information for application elements. The **Application Explorer** retrieves the data using this service API.

REST endpoint is hosted at `http://[baseURI]/Metadata`.

Currently, this endpoint provides metadata for the following elements:

Labels: This gets labels from the system. They have a dual pair key, language, and ID, so that you can retrieve the value of the label.

Example:

```
https://[baseURI]/$metadata/Labels(
  Id='@SVC_ODataLabelFile:Label1’,Language='en-us')
```

Data entities: This returns a JSON-formatted list of all the data entities in the system.

Example:

```
https://[baseURI]/$Metadata/DataEntities
```

These details are shown on Microsoft's official website.

Consuming custom services in SOAP

Dynamics 365 for Finance and Operations also allows us to consume web services on the SOAP endpoint. The main advantages of using SOAP protocol is its descriptive definition, which helps us identify the contracts, proxy classes, and service methods.

Usually, the SOAP endpoint is deployed at `https://<host_uri>/soap/Services/<service_group_name>`.

In this recipe, we will consume our same custom web service at the SOAP endpoint.

Getting ready

In this recipe, we will get the balance of a customer and verify that it is the same in Dynamics 365 for Finance and Operations. Our code is inspired by solutions at `https://github.com/Microsoft/Dynamics-AX-Integration/tree/master/ServiceSamples`. So, we will use the **Soap Utility** solution provided by Microsoft to get our SOAP endpoint address and binding element. Get the help of the NuGet Package Manager Console to install the `Microsoft.IdentityModel.Clients.ActiveDirectory` library.

How to do it...

1. Create a console application and name it `PacktSOAPApplication`. Add a reference to the project `SOAPUtility` project and our authentication project.
2. Add a service reference to our custom web service hosted at the SOAP endpoint `https://d365devdpkcdfe0b0*******os.cloudax.dynamics.com/soap/services/packtCustBalanceServices`.
3. Use the following namespaces in our class to utilize already existing APIs:

```
using AuthenticationUtility;
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;
using SoapUtility.CustBalance;
```

4. Add the following code in our `Program.cs` class, which will call Customer balance web services to get the customer balance from Dynamics 365 for Finance and Operations:

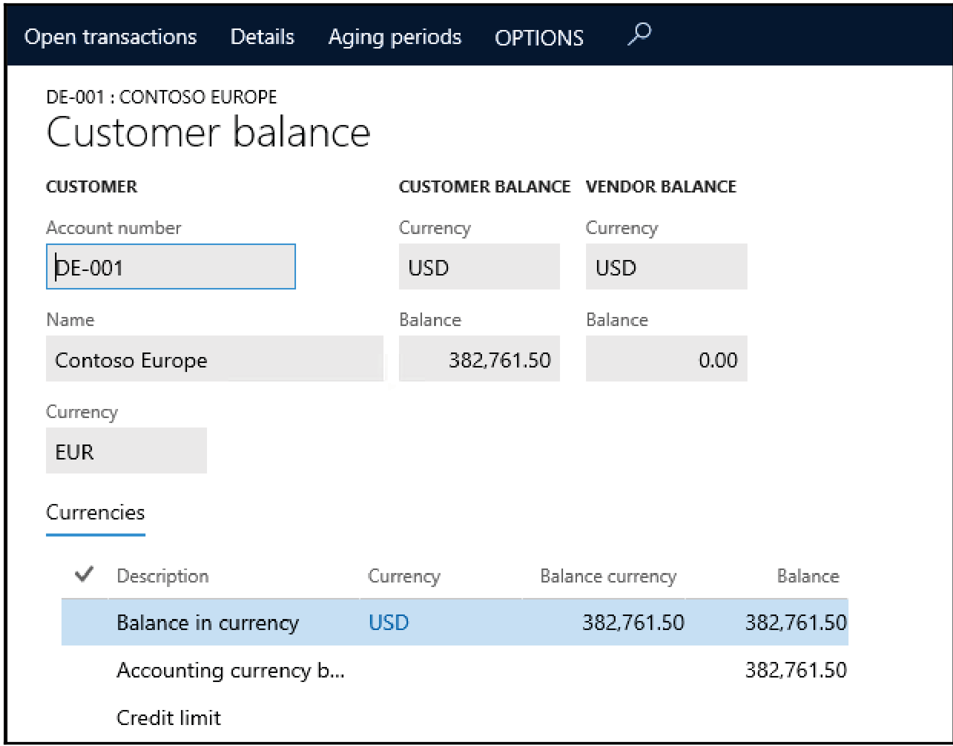
```
class Program
{
  public const string CustBalanceServiceName =
    "packtCustBalanceServices";

  [STAThread]
  static void Main(string[] args)
  {
    var aosUriString = ClientConfiguration.Default.UriString;

    var oauthHeader = OAuthHelper.GetAuthenticationHeader();
    var serviceUriString =
      SoapUtility.SoapHelper.GetSoapServiceUriString(
        CustBalanceServiceName, aosUriString);

    var endpointAddress = new
      System.ServiceModel.EndpointAddress(serviceUriString);
    var binding = SoapUtility.SoapHelper.GetBinding();

    var client = new PacktCustBalanceServiceClient(binding,
     endpointAddress);
    var channel = client.InnerChannel;

    SoapUtility.CustBalance.PacktCustBalanceDataContract
     change;
    change = new
```

```
         SoapUtility.CustBalance.PacktCustBalanceDataContract();
        change.CustomerAccount = "DE-001";
        change.Company = "USMF";
        SoapUtility.CustBalance.processData update;
        var callcontext = new
         SoapUtility.CustBalance.CallContext();
        callcontext.Company = "USMF";
        callcontext.Language = "en-us";
        callcontext.PartitionKey = "initial";
        update = new SoapUtility.CustBalance.processData();
        update._custBalanceDataContract = change;
        update.CallContext = callcontext;
        using (OperationContextScope operationContextScope =
         new OperationContextScope(channel))
        {
          HttpRequestMessageProperty requestMessage =
            new HttpRequestMessageProperty();
          requestMessage.Headers[OAuthHelper.OAuthHeader] =
            oauthHeader;
          OperationContext.Current.OutgoingMessageProperties[
            HttpRequestMessageProperty.Name] = requestMessage;
          var result = (
            (SoapUtility.CustBalance.PacktCustBalanceService)
             channel).processData(update);
            Console.WriteLine("Balance of customer {0} is: {1}",
             change.CustomerAccount, result.result);
        }
        Console.ReadLine();
      }
    }
```

5. Now, to check the output of our code, set our project as a startup project and run the console application by clicking the **Start** button on the Visual Studio toolbar.
6. You will get a prompt to enter your username and password to get an authentication header from azure. After successful authentication, a web service call will be made by passing a contract and proxy to the service method:

```
Balance of customer DE-001 is: 382761.5
_
```

7. This is the output that we get from Dynamics 365 for Finance and Operations web service. Let's verify the output by navigating to **USMF** | **AccountsReceivables** | **Customers** | **AllCustomers**. On customer `DE-001`,press the **Balance** button and you will get a screen as follows:



You can see in the Dynamics 365 Customer form, the balance for this customer is `382,761.50`.

How it works...

Here, we first add a service reference of the SOAP endpoint. After adding a SOAP endpoint, you can use Object browser to view the contract and service clients generated. We utilize `SoapUtility` here to get a formatted endpoint address and binding to call the web service client. We get the authentication packet and specify it on the request header of our web service.

We create an object of data contract to specify the parameters required to execute our business logic. Also, call context here is required to set company, partition key, user ID, and language to the web service. We specify the call context and data contract on our web service method, request a call using the client proxy and get a response from the web service as a customer balance which we print onscreen.

Consuming OData services

In Dynamics 365 for Finance and Operations, OData is a standard protocol for creating and consuming data. Odata provides a **REST** (**Representational State Transfer**) endpoint, for **CRUD** (**Create, Read, Update, and delete**) operations. To expose data entities on OData endpoint, we need to mark the `IsPublic` property of data entity as true in the **Application Object Tree** (**AOT**). Users can use OData to insert and retrieve data using CRUD functionality.

OData doesn't require a call context values, unlike SOAP, by default; it returns only data that belongs to the user's default company. We could specify a *cross-company =true* query option to facilitate the user to fetch data from all other companies that they have access to.

Example:

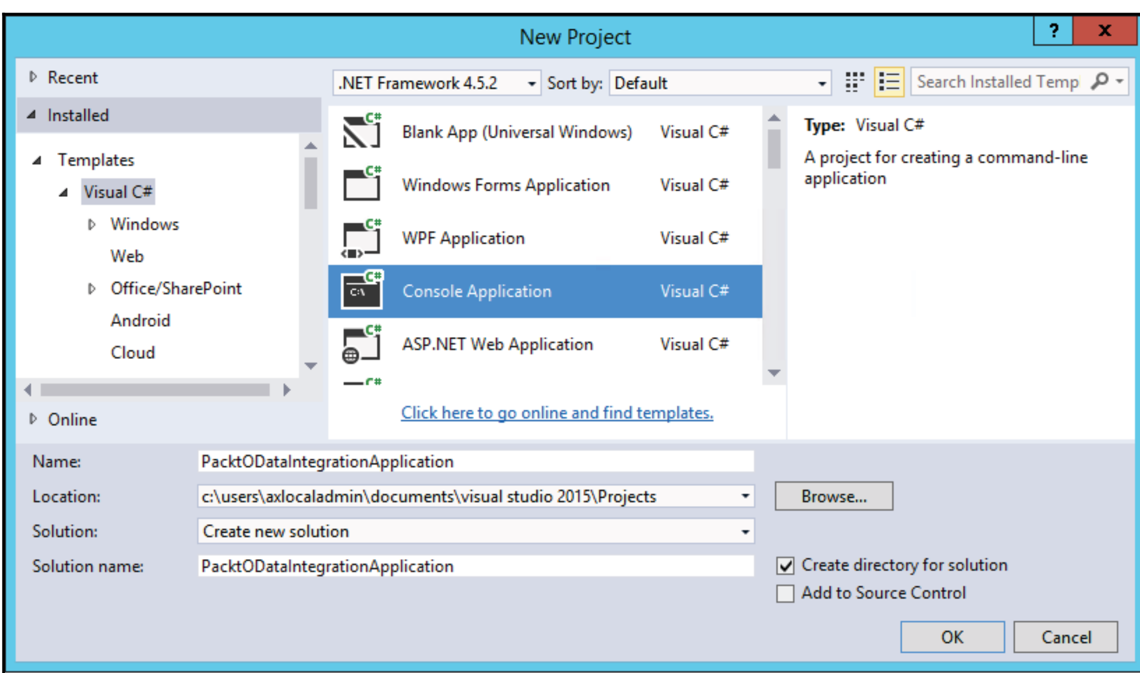
```
http://[baseURI]/data/Customers?cross-company=true
```

Getting ready

In this recipe, we will integrate a Vendor and verify that it is created in Dynamics 365 for Finance and Operations. Our code is inspired by a solution at `https://github.com/Azure-Samples?utf8=%E2%9C%93&query=active-directory`. So, we will use the ODataUtility solution provided by Microsoft for our OData integration, which enables you to get Dynamics 365 data entities into your code.

How to do it...

1. Create a new Console application project and name it as `PacktODataIntegrationApplication`:



2. Add references to your Authentication project, which we created in the earlier recipe and `ODataUtility` and add the following namespaces:

```
using AuthenticationUtility;
using Microsoft.OData.Client;
using ODataUtility.Microsoft.Dynamics.DataEntities;
using System;
using System.Linq;
```

3. In the following code, for the class `ODataCreateVendor` ,we first get an authentication header and set it on the OData context:

```
class ODataCreateVendor
{
  public static string ODataEntityPath =
    ClientConfiguration.Default.UriString + "data";
}
```

4. Add a method to create a new vendor, with the following code. Here, we are using static values for a new vendor:

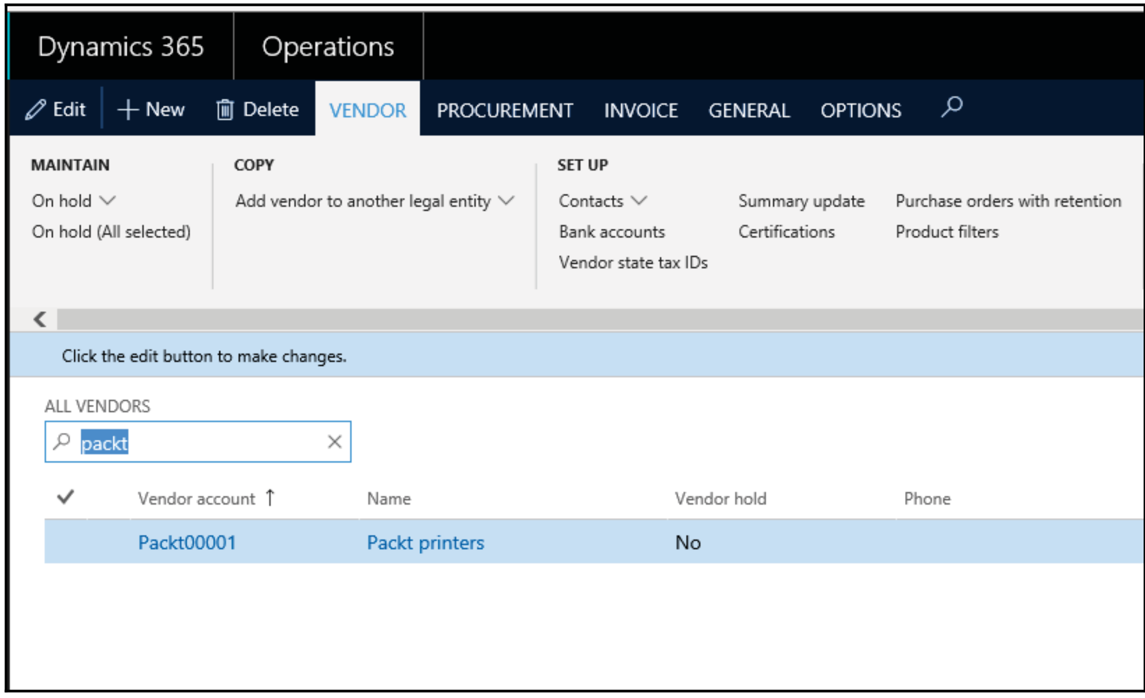
```
public static void CreateVendor(Resources context)
{
  string vendorAccountNumber = "Packt00001";
  try
  {
    Vendor vendorEntity = new Vendor();
    DataServiceCollection<Vendor> vendorEntityCollection =
      new DataServiceCollection<Vendor>(context);
    vendorEntityCollection.Add(vendorEntity);
    vendorEntity.VendorAccountNumber = vendorAccountNumber;
    vendorEntity.VendorName = "Packt printers";
    vendorEntity.VendorSearchName = "Packt printers";
    vendorEntity.VendorPartyType = "Organization";
    vendorEntity.AddressCountryRegionId = "IND";
    vendorEntity.CurrencyCode = "INR";
    vendorEntity.VendorGroupId = "10";
    vendorEntity.DataAreaId = "USMF";
    context.SaveChanges(
      SaveChangesOptions.PostOnlySetProperties |
       SaveChangesOptions.BatchWithSingleChangeset);
    Console.WriteLine(string.Format("Vendor {0} - created !",
     vendorAccountNumber));
  }
  catch (DataServiceRequestException e)
  {
    Console.WriteLine(string.Format("Vendor {0} - failed !",
     vendorAccountNumber));
  }
}
}
```

5. Now, add a main method with the following code, where we set the other required parameter and get the authentication header:

```
static void Main(string[] args)
{
  Uri oDataUri = new Uri(ODataEntityPath, UriKind.Absolute);
  var context = new Resources(oDataUri);

  context.SendingRequest2 += new
   EventHandler<SendingRequest2EventArgs>(
     delegate (object sender, SendingRequest2EventArgs e)
   {
     var authenticationHeader =
       OAuthHelper.GetAuthenticationHeader();
     e.RequestMessage.SetHeader(OAuthHelper.OAuthHeader,
      authenticationHeader);
   });
   CreateVendor(context);
   Console.ReadLine();
}
```

6. Set our project as a startup project and execute the project using the **Start** button in Visual Studio. You will get a prompt to enter a username and password, which you need to enter to get authentication. Once authenticated, our OData application will create the user and show the following screen:

7. We could verify the vendor created by logging into Dynamics 365 for Finance and Operations USMF company and opening **ALL VENDORS**:



As you can see, vendor `Packt00001` has been added in Dynamics 365 successfully.

How it works...

OData lets you interact with data by using RESTful web services. Adding using `ODataUtility.Microsoft.Dynamics.DataEntities` enables you to get access to Dynamics 365 data entities in C# code. Now, in the `createVendor` method, assign values to all respective fields in the data entity. Make sure all related masters, such as `VendorGroupId` and `vendorPartyType` already exist in the Dynamics 365 database.

Finally, in the `main` method, get an authentication header using:

```
var authenticationHeader = OAuthHelper.GetAuthenticationHeader()
```

And, on successful authentication, call the `createVendor` method to insert the vendor record in the respective tables in the Dynamics 365 datasource. Finally, call `Console.ReadLine();` to see the result onscreen.

There's more...

In the preceding recipe, we used static vales for vendor creation. You can convert this into your business logic and use it for your actual development, where you get vendors data from an external source and insert it into Dynamics 365 for Finance and Operations.

See also

- For more information about OData, see the following web page: `http://www.odata.org/documentation/`

Consuming external web services

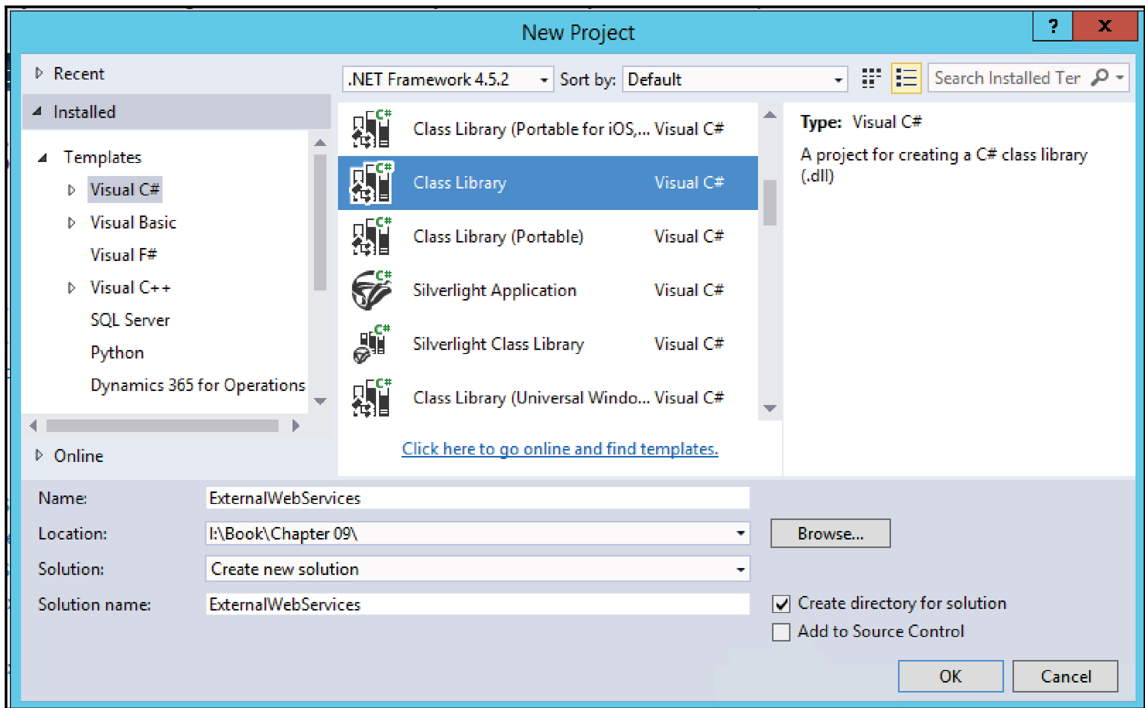
Dynamics 365 for Finance and Operations supports consuming external web services by writing a simple wrapper class on the external web service endpoint in Visual Studio and adding its reference Dynamics 365 for Finance and Operations project.

Getting ready

For this code walkthrough, we will take a free online web service for a currency converter. You can find the WSDL at `http://currencyconverter.kowabunga.net/converter.asmx`

How to do it...

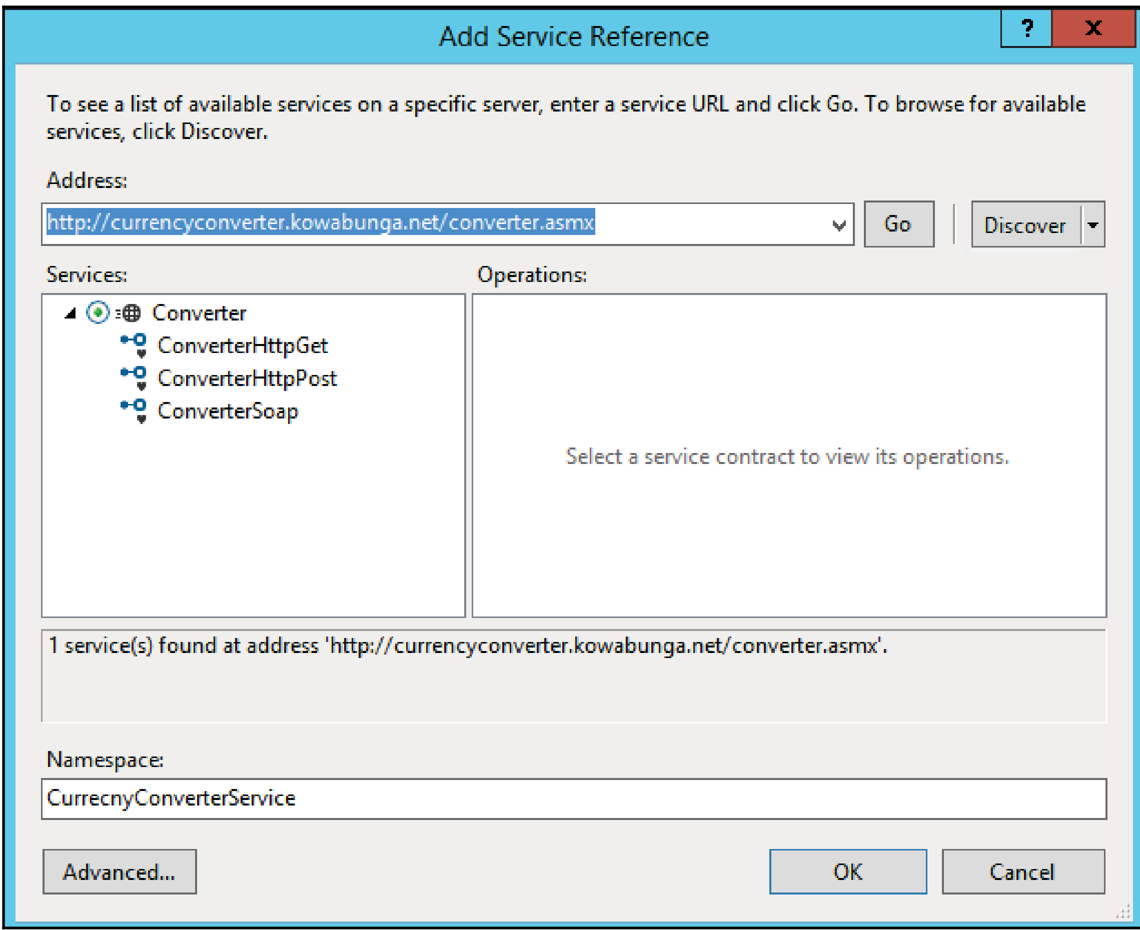
1. Add a new **Class Library** and name it `ExternalWebservices`:



2. Add a service reference under the **References** node of our newly created project. Use the following service `http://currencyconverter.kowabunga.net/converter.asmx`.

   Give a name `CurrencyConverterServices` in the namespace.

3. By clicking on the **Go** button, the system will fetch and show all the available services and operations in this web service, as follows:



Click on the **OK** button.

4. It will add a new node in your project service reference.
5. Check the `app.config` file. It should look as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <bindings>
            <basicHttpBinding>
                <binding name="ConverterSoap" />
            </basicHttpBinding>
            <customBinding>
                <binding name="ConverterSoap12">
                    <textMessageEncoding messageVersion="Soap12" />
                    <httpTransport />
                </binding>
            </customBinding>
        </bindings>
        <client>
            <endpoint address="http://currencyconverter.kowabunga.net/converter.asmx"
                binding="basicHttpBinding" bindingConfiguration="ConverterSoap"
                contract="CurrencyConverterService.ConverterSoap" name="ConverterSoap" />
            <endpoint address="http://currencyconverter.kowabunga.net/converter.asmx"
                binding="customBinding" bindingConfiguration="ConverterSoap12"
                contract="CurrencyConverterService.ConverterSoap" name="ConverterSoap12" />
        </client>
    </system.serviceModel>
</configuration>
```

6. Now, go to the `class1` node and add the following namespace:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using ExternalWebServices.CurrencyConverterService;
using System.ServiceModel;
```

7. Rename the class name to `CurrencyConverter`, and replace its main method with a new method `getCurrecyRate`, as shown in the following code:

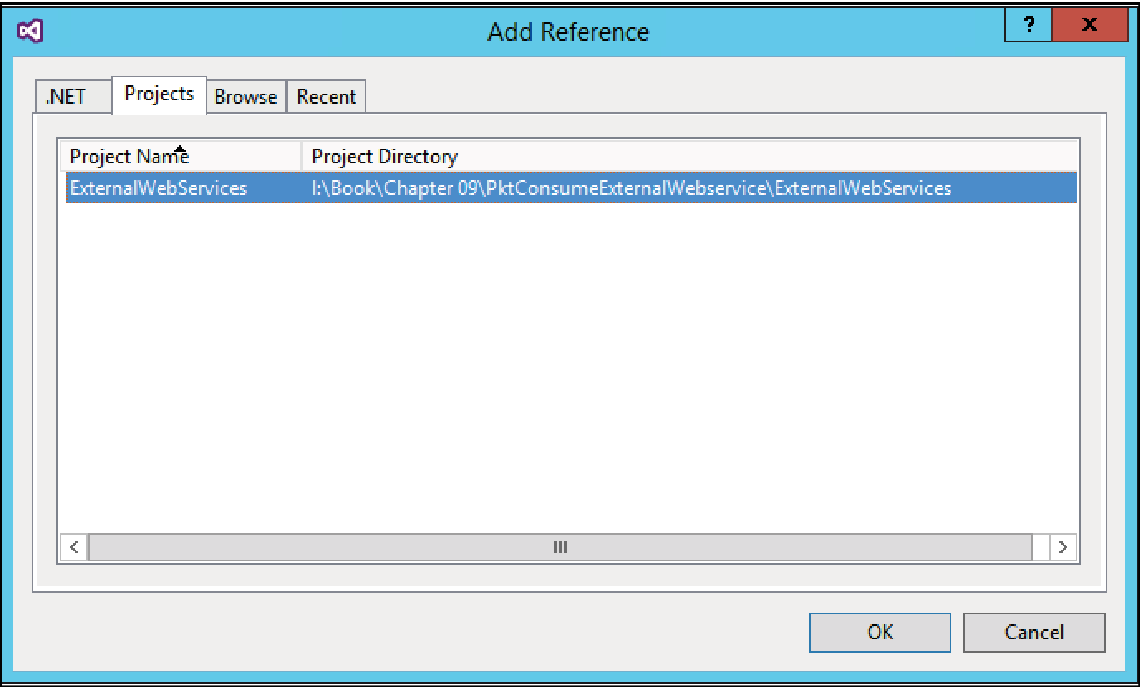
```
public string getCurrencyRate(String cur1, string cur2)
{
  BasicHttpBinding    httpBinding = new BasicHttpBinding();
  EndpointAddress endPoint = new EndpointAddress(
    @"http://currencyconverter.kowabunga.net/converter.asmx");

  var soapClient = new ConverterSoapClient(httpBinding,
   endPoint);
  var usdToinr = soapClient.GetConversionRate(
```

```
        cur1.ToString(), cur2.ToString(),DateTime.Now);
      return usdToinr.ToString();

    }
```

8. Save all your code and build the project.

Sometimes, you may face unexpected errors during a build. In such cases, right-click on the solution and clean the solution. Now, try to rebuild the complete solution.

9. Now, our web service is ready to be consumed in Dynamics 365. Add a new Operation Project in your solution. Name it `PktExternalWebServiceDAX`.
10. In the **Reference** node, right-click and add a new reference. You can choose the project and it will add the `dll` file automatically in the reference node. See the following screenshot:



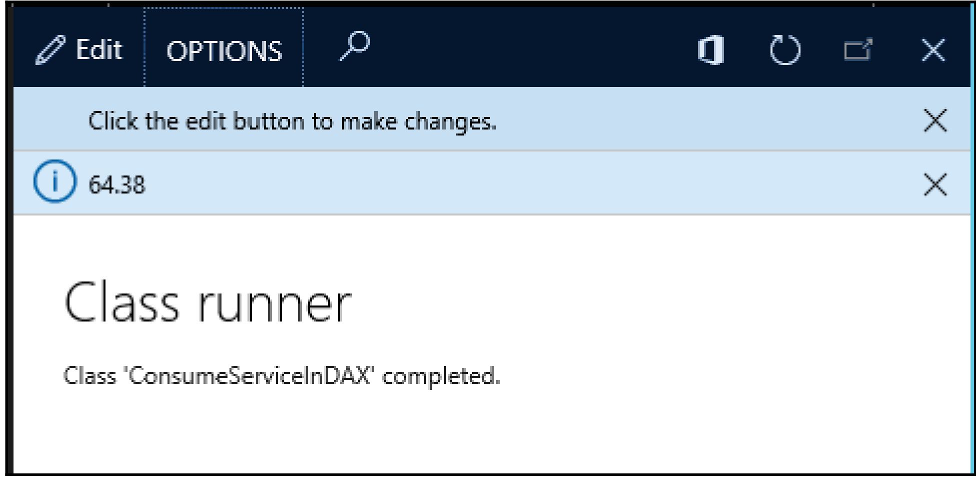
11. Once you click on the OK button, this must be added in the **Reference** node of the current project.
12. Now add a runnable class in your Operation Project, name it `ConsumeServiceInDAX`, and add a namespace of:

    ```
    using ExternalWebServices;
    ```

13. Add the following code in the main method:

    ```
    ExternalWebServices.currencyConverter currencyCoverter =
      new ExternalWebServices.currencyConverter();
    var convRate = str2Num(currencyCoverter.getCurrencyRate("USD",
      "INR"));
    info(strFmt("%1", convRate));
    ```

14. Save all your code and build the solution. Set this project `PktExternalWebServiceDAX` as a startup project and class `ConsumeServiceInDAX` it as a startup Object.
15. To test the code, run the solution now. You will get your conversion rate in the following screen:

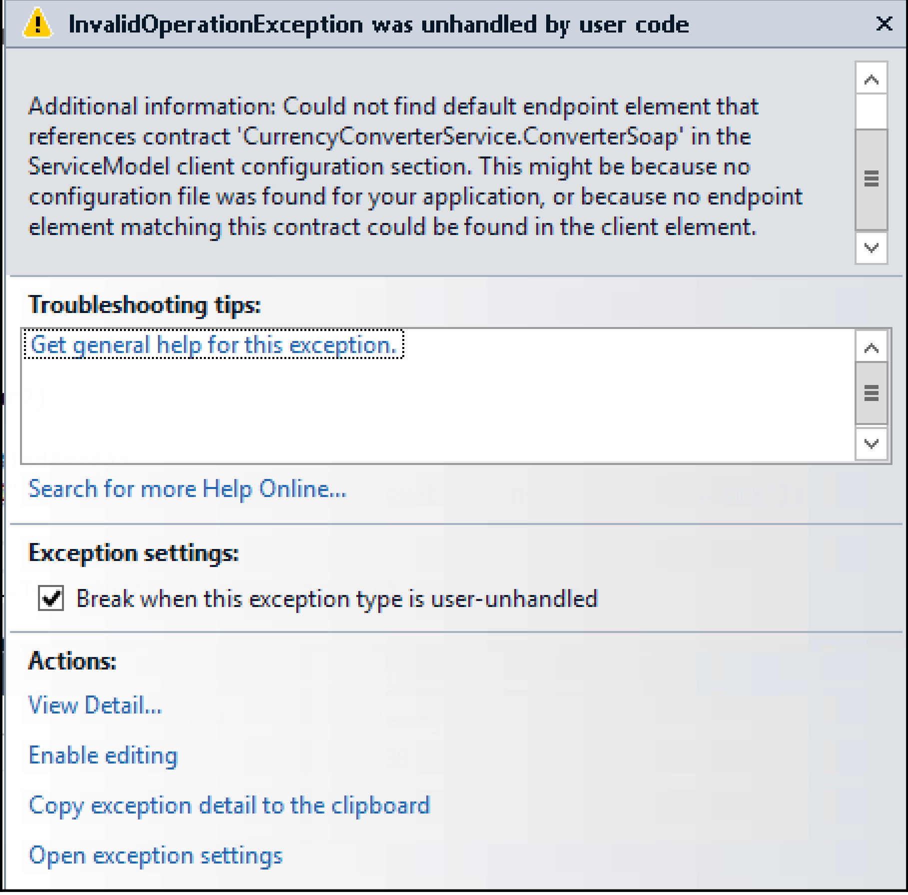


`64.38` is the currency conversion rate for USD and INR.

How it works...

Every web service comes with certain services and operations; you can get all these details along with the web service. The very first thing we added was a service reference to our C# project that enables us to interact with this web service and use its methods/services in code. After adding a service endpoint, you can use Object browser to view the contract and service clients generated.

Next, you have to create a binding and endpoint for this service; without these you may get the following error while it is running:



C# code doesn't take a string directly; you have to convert all objects into a string using the `ToString()` method.

On the other side, in a Dynamics 365 project, once you add a C# project in the **Reference** node, you can interact with its class now. Adding a namespace on top of your class will help to reduce extra lines of code.

Lastly, create an object of C# service; it's the same as creating an object for any other class:

```
ExternalWebServices.currencyConverter currencyCoverter =
  new ExternalWebServices.currencyConverter();
```

And convert the return value, that is, a string into a number to show as output:

```
var convRate = str2Num(currencyCoverter.getCurrencyRate("USD",
 "INR"));
info(strFmt("%1", convRate));
```

There's more...

Browse this service on an internet browser to check the services, description and all available operations in this service:
`http://currencyconverter.kowabunga.net/converter.asmx`

The following are available operations for the preceding service:

- ConvertDataTableColumn
- GetConversionAmount
- GetConversionRate
- GetCultureInfo
- GetCurrencies
- GetCurrencyRate
- GetCurrencyRates
- GetLastUpdateDate

Check each of them to get detailed information on a particular operation code reference.

See also

There are lots of free web services over the internet; you can get them free of cost and use them in your code. Some of them are:

- `http://www.webservicex.net/new/Home/Index`
- `http://www.visualwebservice.com/`

We will not support any website/company/person if they use any web service in their code. Please do study and take cautions when using any web services in your production environment.

10
Improving Development Efficiency and Performance

In this chapter, we will cover the following recipes:

- Using Extension
- Caching a display method
- Calculating code execution time
- Enhancing insert, update, and delete operations
- Writing efficient SQL statements
- Using event handler
- Creating a Delegate method

Introduction

We are entering a new era of Dynamics AX as Dynamics 365 for Finance and Operations, and it is quite common for many small and large Microsoft Dynamics 365 for Finance and Operations installations/implementations to suffer from performance issues. These issues can be caused by insufficient infrastructure, incorrect configuration, ineffective code, and many other reasons.

There are lots of ways to troubleshoot and fix performance issues. Using standard tools or review logic. This chapter discusses a few simple must-know techniques to write code properly and improve logic to deal with basic performance issues. It is hard to get a complete guide to solve performance issues in Dynamics 365 for Finance and Operations.

Using extensions

In Dynamics 365 for Finance and Operations, development requirements can be better achieved using an extension approach on standard application objects. Unlike overlayering in AX 2012, extensions in Dynamics 365 for Finance and Operations don't overlay the base model elements. Instead, it compiles them in separate assemblies, which allows us to customize on standard objects and associated business logic using Event Handler. Extensions provide minimal overhead when a base package is upgraded.

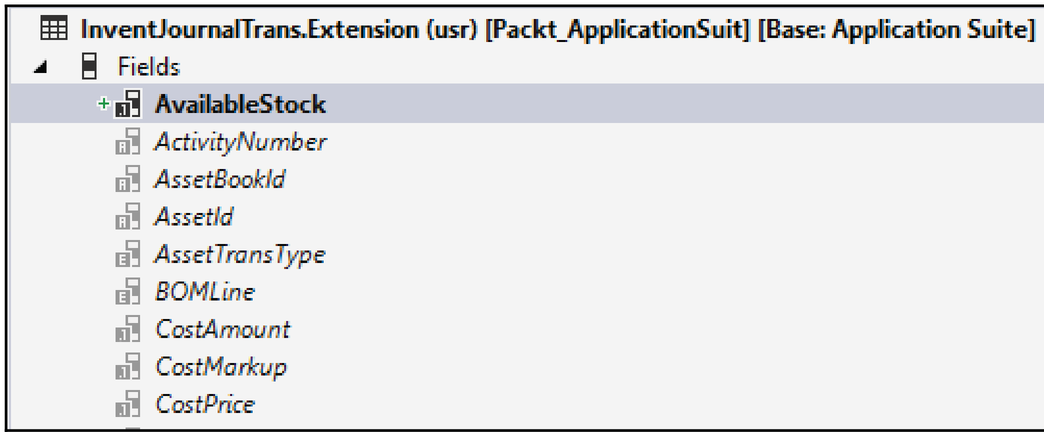
In this recipe, we will be extending `InventJournalTrans` and adding new fields in it.

How to do it...

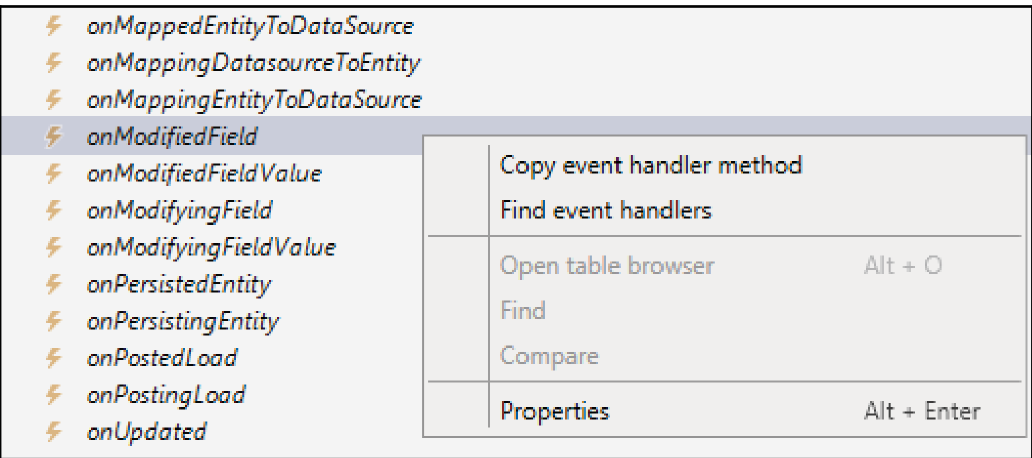
1. Create a new project in Visual Studio `UsingExtensions` and select package, which references `Application` suit package.
2. In **Application** explorer, find the table `InventJournalTrans` and create its extension.
3. Create a new field, `AvailableStock`, with the following properties:

Property	Value
Name	`AvailableStock`
Label	`Available stock`
EDT	Qty

4. Your new field should look as follows:



5. Copy its event handler method `onModifiedField`:

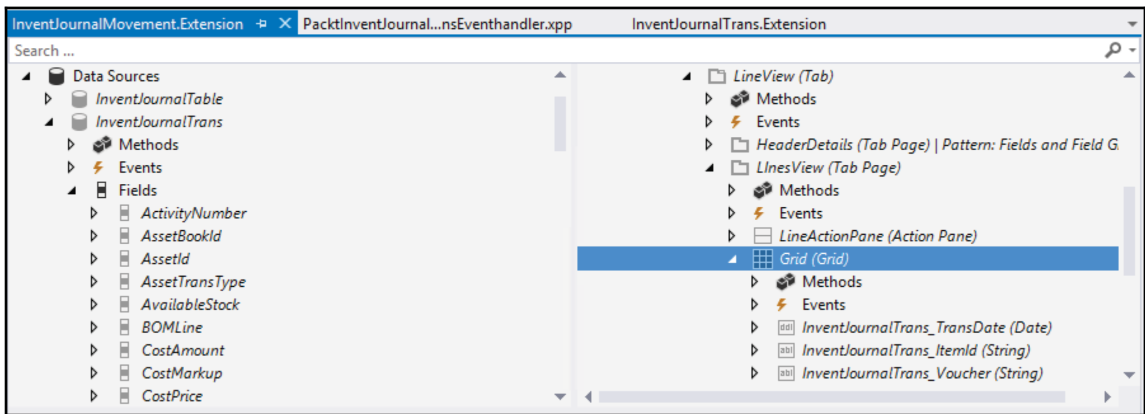


6. Create a new class, `PacktInventJournalTransEventhandler`, and paste the `onModifiedField` event handler method in this class. In addition to the default code, we need to get the `ModifyFieldEventArgs` arguments to get the field ID and use switch case. The `DataEventHandler` specifies the source of the event and `DataEventType` is an enum for the type of event supported. Once the event arguments are received, we get field ID to call our logic based on a switch-case ladder. Modify the code in this class so that it looks as follows:

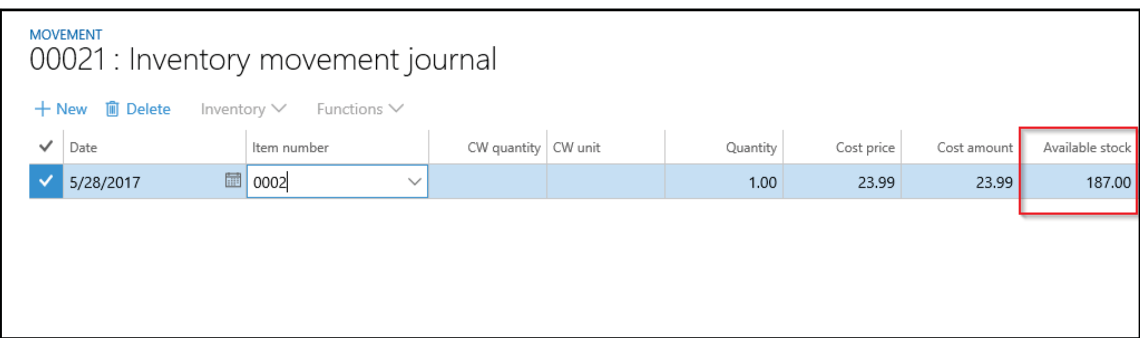
```
class PacktInventJournalTransEventhandler
{
  /// <summary>
```

```
///
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
[DataEventHandler(tableStr(InventJournalTrans),
  DataEventType::ModifiedField)]
public static void InventJournalTrans_onModifiedField
   (Common sender, DataEventArgs e)
{
  ModifyFieldEventArgs args = e as ModifyFieldEventArgs;
  InventJournalTrans inventJournalTrans = sender
   as InventJournalTrans;

  switch(args.parmFieldId())
  {
    case fieldNum(InventJournalTrans,ItemId):
    inventJournalTrans.AvailableStock =
      InventOnhand::newItemId(inventJournalTrans.ItemId)
      .availPhysical();
    break;
  }
}
}
```

7. In **Application** explorer, find the form `InventJournalTrans` and create its extension. Expand the data sources, drag field from `InventJournalTrans` data source to Grid in the **LineView(Tab)** page. It should look as follows:



8. Navigate to **Inventory Management** | **Journal entries** | **Items** | **Movement** and create a `movement journal`. Go into journal lines and select item. Our code will fetch the available physical quantity of items and display it in the field:



How it works...

Here, we create a new field in `InventJournalTrans.Extension` that extends the base package element in our custom package. We copy the event handler of the method on `ModifiedField` and copy it in our custom class, where we modify it according to our needs. As we know, we have to fetch the stock of an item when it is changed, so we have put our code on modified events on the items `id` field. We fetch the stock using the `InventOnHand` class method `availPhysical()` and save it in our field. This way, we have customized application code without overlayering the base package application objects. This approach helps in easy upgrades in base application objects such as classes, tables, and forms. Event handlers on table methods are already provided by the system and they can be utilized.

Always use proper comments in between code. Don't put any single lines of code without comments. Always use a prefix for your new method, new fields, and objects. Normally, we use the initial three letters for this, such as `pkt` for PacktPub.

Caching a display method

In Dynamics 365 for Finance and Operations, `display` methods are still used in some places to show additional information on forms or reports such as fields, calculations, and more. Although they are shown as physical fields, their values are a result of the methods to which they are bound.

The `display` methods are executed each time the form is redrawn. This means that the more complex the method is, the longer it will take to display it on the screen. Hence, it is always recommended to keep the code in the `display` methods to a minimum.

The performance of the `display` methods can be improved by caching them. This is when a `display` method's return value is retrieved from a database or calculated only once and subsequent calls are made to the cache.

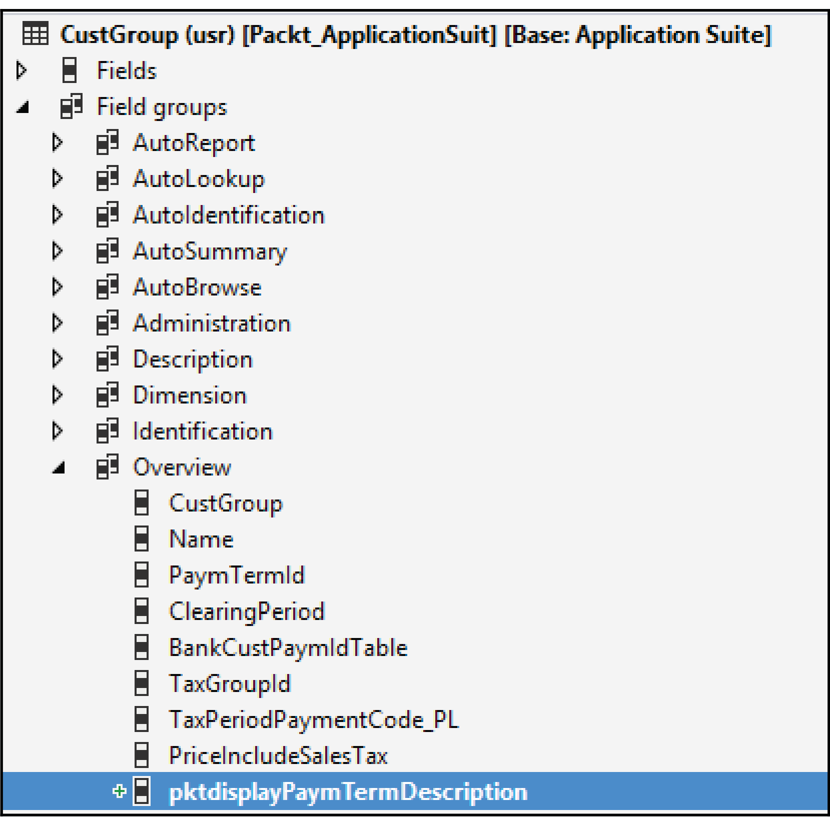
In this recipe, we will create a new cached `display` method for the PaymentTerm description. We will also discuss a few scenarios in order to learn how to properly use caching.

How to do it...

1. In the **Application** explorer, locate the `CustGroup` table and customize it into your project.
2. Add a `display` method with the following code snippet:

```
/// <summary>
///New method added by YourName for req:PacktChapter10
///display description for Payment term
/// </summary>
display Description pktdisplayPaymTermDescription()
{
    return (select firstOnly Description from PaymTerm
    where PaymTerm.PaymTermId == this.PaymTermId).Description;
}
```

3. Add the newly created method to the table's `Overview` group, right beneath the `PaymTermId` field, so it will update it on form design on the `Overview` field group and we don't need to add this field on the form separately.

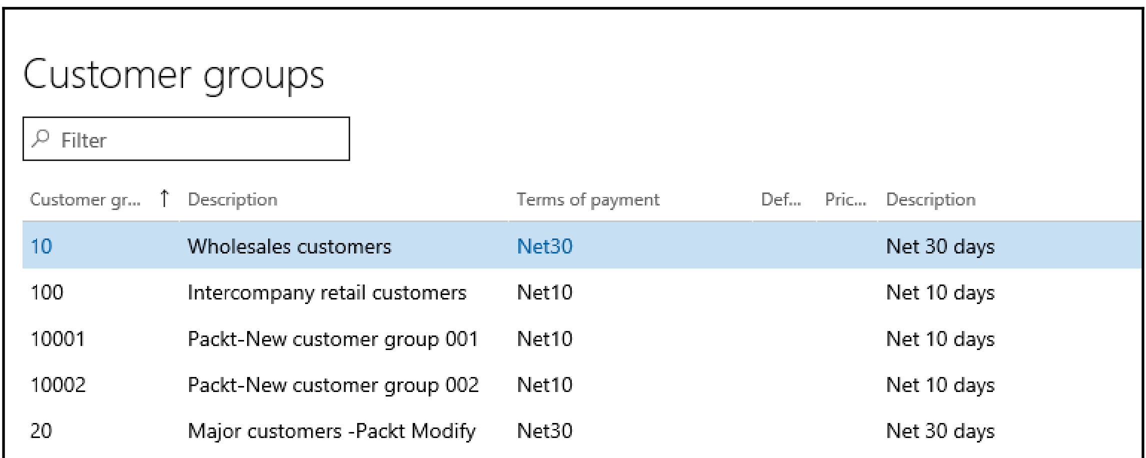


4. In the Application Object Tree, find the `CustGroup` form and add it into your project.
5. Override the `init()` method of its `CustGroup` data source with the following code snippet:

```
/// <summary>
/// Added by yourname on May 13 for purose
/// </summary>
public void init()
{
  super();
  this.cacheAddMethod(tableMethodStr(CustGroup,
     pktdisplayPaymTermDescription));
}
```

6. To test the `display` method, **Save and build** your solution.

7. Navigate to **Accounts receivable** | **Setup** | **Customers** | **Customer groups**.
8. You will see a newly added display method showing **Terms of Payment** in the form, as shown in the following screenshot:



How it works...

In this recipe, we created a new `display` method in the `CustGroup` table to show the description of **Terms of Payment**, which is defined in a **Customer group** record. In the method, we use a query to retrieve only the `Description` field from the `PaymTerm` table. Here, we can use the `find()` method of the `PaymTerm` table, but that will have decreased the `display` method's performance, as it returns the whole `PaymTerm` record when we only need a single field. In a scenario such as this, when there are only a few records in the table, it is not so important; however, in the case of millions of records, the difference in the performance will be noticeable. This is precisely why we need to cache the display method that we have covered in this recipe. We also add the method that we created to the **Overview** group in the table in order to ensure that it automatically appears on the **Overview** screen of the **Customer group** form.

In order to cache the `display` method, we override the `init()` method of the `CustGroup` data source and call it's `cacheAddMethod()` method to ensure that the method's return values are stored in the cache.

The `cacheAddMethod()` method instructs the system's caching mechanism to load the method's values into the cache for the records visible on the screen, plus some subsequent records. It is important that only the display methods that are visible in the **Overview** tab are cached. The display methods located in different tabs can show a value from one record at a time, even though the caching mechanism loads values from multiple records into the cache.

Speaking about the `display` method caching, there are other ways to do this. One of the ways is to place the `SysClientCacheDataMethodAttribute` attribute at the top of the `display` method, as shown in the following code snippet:

```
[SysClientCacheDataMethodAttribute]
display Description displayPaymTermDescription()
{
   return (select firstOnly Description from PaymTerm
   where PaymTerm.PaymTermId == this.PaymTermId).Description;
}
```

In this case, the method will automatically be cached on any form where it is used without any additional code.

Another way is to change the `CacheDataMethod` property of the form's control to `Yes`. This will have the same effect as using the `cacheAddMethod()` method or the `SysClientCacheDataMethodAttribute` attribute.

There's more...

Display methods can be used in Lookup methods as well. However, you will not be able to perform any sorting and filtering in this field. Using the following code syntax, you can add a new field in Lookup using a display method:

```
sysTableLookup.addLookupMethod(tableMethodStr(TableName, MethodName));
```

You can add this in your Lookup method right after all your physical fields.

Calculating code execution time

When working on improving existing code, there is always the question of how to measure results. There are numerous ways to do this, for example, visually assessing improvements, getting feedback from users, using the code profiler and/or trace parser, and various other methods.

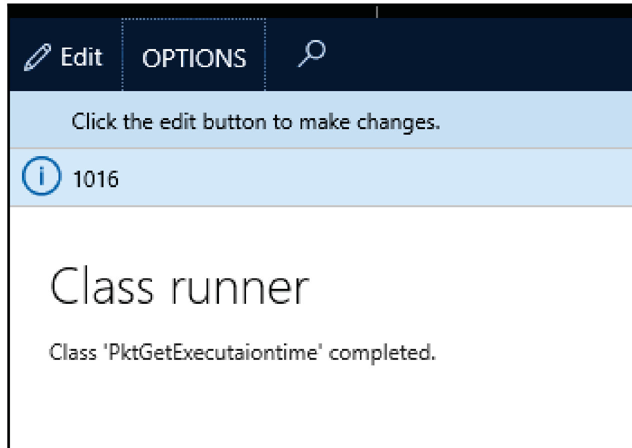
In this recipe, we will discuss how to measure the code execution time using a very simple method, just by temporarily adding a few lines of code. In this way, the execution time of the old code can be compared with that of the new one in order to show whether any improvements were made.

How to do it...

1. In the solution, add a new runnable class and write the following code snippet:

```
class PktGetExecutaiontime
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    int start;
    int end;

    start = WinAPI::getTickCount();
    sleep(1000); // pause for 1000 milliseconds
    end = WinAPI::getTickCount();

    info(strFmt("%1", end - start));

  }
}
```

2. Set this class as a startup object in **project** and **run** the solution to see how many milliseconds it takes to execute the code, as shown in the following screenshot:



How it works...

In this recipe, the `sleep()` command simulates our business logic, which measures the execution time.

The main element is the `getTickCount()` method of the standard `WinAPI` class. The method returns the `TickCount` property of the .NET environment, which is a 32-bit integer containing the amount of time, in milliseconds, that has passed since the last time the computer was started.

We place the first call to the `getTickCount()` method before the code we want to measure, and we place the second call right after the code. In this way, we know when the code was started and when it was completed. The difference between the times is the code execution time, in milliseconds.

Normally, using such a technique to calculate the code execution time does not provide useful information, as we cannot exactly tell whether it is right or wrong. It is much more beneficial to measure the execution time before and after we optimize the code. In this way, we can clearly see whether any improvements were made.

There's more...

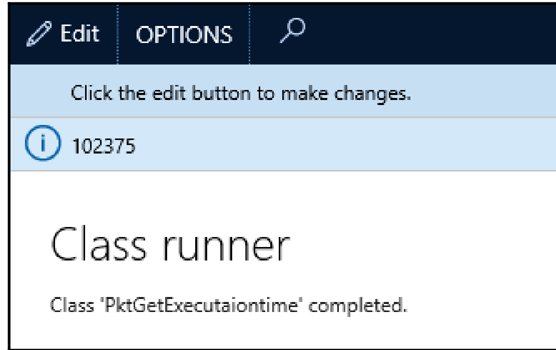
The approach described in the previous section can be successfully used to measure long-running code, such as numerous calculations or complex database queries. However, it may not be possible to assess code that takes only a few milliseconds to execute.

The improvement in the code may not be noticeable, as it can be greatly affected by the variances caused by the current system conditions. In such a case, the code in question can be executed a number of times so that the execution times can be properly compared.

To demonstrate this, we can modify the previously created job as follows:

```
class PktGetExecutaiontime
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    int i;
    int64 timeTaken;
    System.DateTime dateTimeNow;
    System.TimeSpan         timeSpan;
    utcdatetime             startDateTimeUTC;

    startDateTimeUTC = DateTimeUtil::utcNow();
    for (i = i; i <= 20; i++)
    {
      sleep(1000); // pause for 1000 milliseconds
    }
    dateTimeNow       =   System.DateTime::get_UtcNow();
    timeSpan          =
      dateTimeNow.Subtract(utcDateTime2SystemDateTime
       (startDateTimeUTC));
    timeTaken         =   timeSpan.get_TotalMilliseconds();

    info(strFmt("%1", timeTaken));
  }
}
```

Your output must look like the following line:



Now, the execution time will be much longer, as we add a `for` loop to increase the execution time and, therefore, it is easier to assess.

Enhancing insert, update, and delete operations

Dynamics 365 for Finance and Operations is a three-tier architecture and it takes a significant amount of time for a database call to insert, update, and delete. The system provides us with some constructs that allow us to insert/update/delete more than one record into the database in a single trip, which reduces communication between the application and the database and it increases performance.

In this recipe, we will use these constructs and see how to do so effectively.

How to do it...

1. Create a custom table `PacktCustomerInvoices` where we will be inserting records with the following fields:

   Field: `InvoiceAccount`

Property	Value
Name	`InvoiceAccount`

Label	`Invoice account`
EDT	`AccountNum`

Field: `InvoiceId`

Property	**Value**
Name	`InvoiceId`
Label	`Invoice id`
EDT	`DocumentNum`

Field: `InvoiceAmount`

Property	**Value**
Name	`InvoiceAmount`
Label	`Invoice amount`
EDT	`Amount`

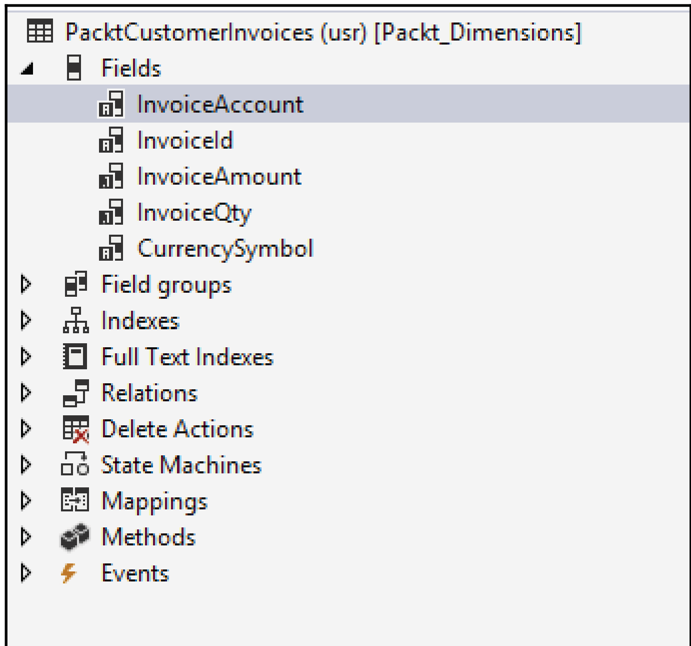
Field: `InvoiceQty`

Property	**Value**
Name	`InvoiceQty`
Label	`Invoice qty`
EDT	`Qty`

Field: `CurrencySymbol`

Property	**Value**
Name	`CurrencySymbol`
Label	`Currency`
EDT	`CurrencySymbol`

After all the preceding fields, your table will look as follows:

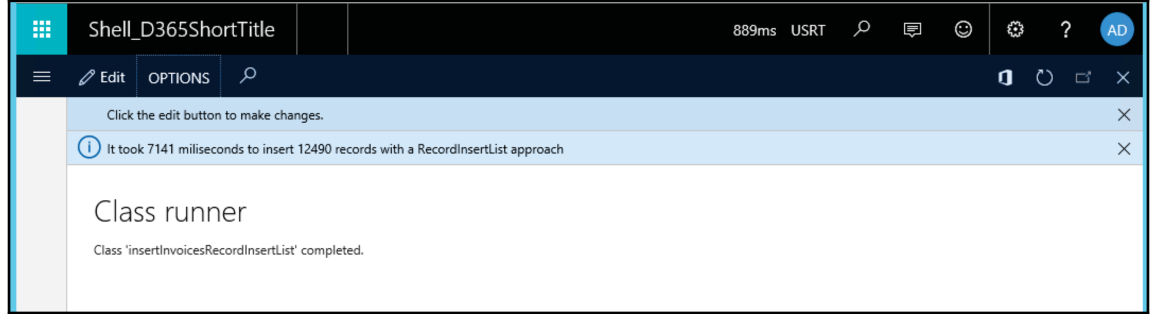


2. Create a new runnable class `insertInvoicesRecordInsertList` and add the following code:

```
class insertInvoicesRecordInsertList
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    CustInvoiceJour           custInvoiceJour;
    PacktCustomerInvoices     customerInvoice;
    int64                     timeTaken;
    System.DateTime           dateTimeNow;
    System.TimeSpan           timeSpan;
    utcdatetime               startDateTimeUTC;
    int                       i;
     // This collection will store the records that must be
     inserted into the database, we provide skipInsertMethod and
     skipDatabaseLog as true and rest parameters as false
     RecordInsertList customerInvoicesToBeInserted = new
```

```
        RecordInsertList(tableNum(PacktCustomerInvoices), true,
        true, true, false, false,customerInvoice);

       //Initialize  the start date time.
       startDateTimeUTC = DateTimeUtil::utcNow();

      //Select only the required fields from the table
      while select InvoiceAccount,InvoiceId,InvoiceAmount,Qty,
       CurrencyCode from custInvoiceJour
    {
      // Initializing the buffer with invoice records
     customerInvoice.InvoiceAccount=custInvoiceJour.InvoiceAccount;
     customerInvoice.InvoiceI = custInvoiceJour.InvoiceId;
     customerInvoice.InvoiceAmount = custInvoiceJour.InvoiceAmount;
     customerInvoice.InvoiceQt = custInvoiceJour.Qty;
     customerInvoice.CurrencySymbol = custInvoiceJour.CurrencyCode;

     // Instead of inserting the record into the database, we
       will add it to the RecordInsertList array
     customerInvoicesToBeInserted.add(customerInvoice);
     i++;
    }
     // After fulfilling the array with the elements to
     be inserted we are
     // read to execute the insert operation
     customerInvoicesToBeInserted.insertDatabase();
     dateTimeNow       =   System.DateTime::get_UtcNow();
     timeSpan          =
       dateTimeNow.Subtract(utcDateTime2SystemDateTime
        (startDateTimeUTC));
     timeTaken         =   timeSpan.get_TotalMilliseconds();

     info(strFmt('It took %1 miliseconds to insert %2 records with
       a RecordInsertList approach', timeTaken,i));
    }
  }
```

3. The Infolog shows us `12490` records inserted in the table in `7141` milliseconds using the `RecordInsertList` approach:

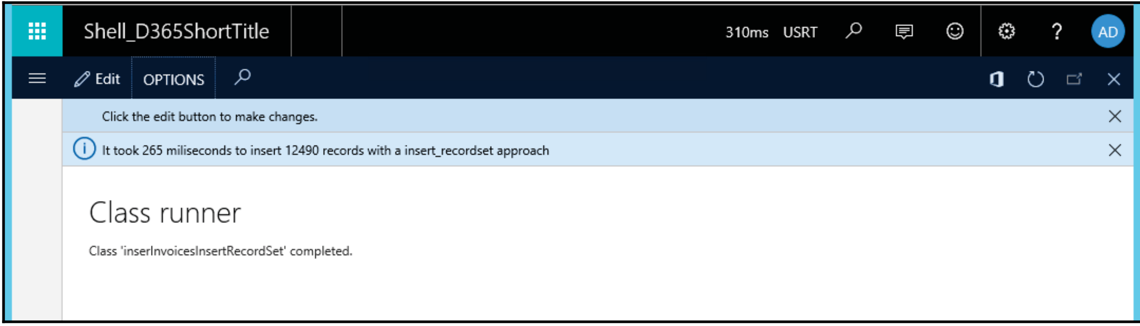


4. Furthermore, we could enhance performance by using `insert_RecordSet`. Let's create a new class `inserInvoicesInsertRecordSet` and add the following code:

```
class insertInvoicesInsertRecordSet
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    CustInvoiceJour          custInvoiceJour;
    PacktCustomerInvoices    customerInvoice;
    int64                    timeTaken;
    System.DateTime          dateTimeNow;
    System.TimeSpan          timeSpan;
    utcdatetime              startDateTimeUTC;
    int64                    i;

    select count(RecId) from custInvoiceJour;
    i = custInvoiceJour.RecId;

    startDateTimeUTC = DateTimeUtil::utcNow();

    insert_recordset
      customerInvoice(InvoiceAccount,InvoiceId,InvoiceAmount,
        InvoiceQty, CurrencySymbol)
      select
    InvoiceAccount,InvoiceId,InvoiceAmount,Qty,CurrencyCode
        from custInvoiceJour;
    dateTimeNow        =   System.DateTime::get_UtcNow();
    timeSpan           =
      dateTimeNow.Subtract(utcDateTime2SystemDateTime
        (startDateTimeUTC));
```

```
            timeTaken         =   timeSpan.get_TotalMilliseconds();

            info(strFmt('It took %1 miliseconds to insert %2 records
              with a insert_recordset approach', timeTaken,i));
        }
    }
```

5. The Infolog shows us `12490` records inserted in the table in `265` milliseconds using the `insert_RecordSet` approach:



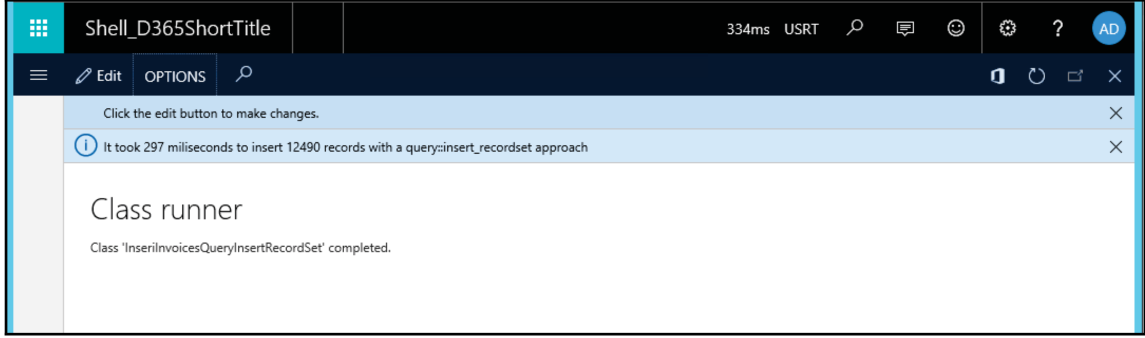
6. We could also use the `query::insert_recordset` method for improved performance, which could help us in different scenarios. Let's perform the same task using this approach. Create a new class `InseriInvoicesQueryInsertRecordSet` and add the following code:

```
class InsertInvoicesQueryInsertRecordSet
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    Map fieldMapping;
    Query query;
    QueryBuildDataSource    qbdsCustInvoiceJour;
    QueryBuildFieldList     fieldList;
    PacktCustomerInvoices   customerInvoice;
    CustInvoiceJour         custInvoiceJour;
    int64                   timeTaken;
    System.DateTime         dateTimeNow;
    System.TimeSpan         timeSpan;
    utcdatetime             startDateTimeUTC;
    int64                   i;
```

```
//Count the records to be inserted for demo purpose

select count(RecId) from custInvoiceJour;
i = custInvoiceJour.RecId;

startDateTimeUTC = DateTimeUtil::utcNow();

//Prepare the query object
query = new Query();

// Prepare the data source for record set operation
qbdsCustInvoiceJour =
  query.addDataSource(tableNum(CustInvoiceJour));

//Add field selection list
fieldList = qbdsCustInvoiceJour.fields();
fieldList.addField(fieldNum(CustInvoiceJour,
 InvoiceAccount));
fieldList.addField(fieldNum(CustInvoiceJour, InvoiceId));
fieldList.addField(fieldNum(CustInvoiceJour,
 InvoiceAmount));
fieldList.addField(fieldNum(CustInvoiceJour,Qty));
fieldList.addField(fieldNum(CustInvoiceJour,CurrencyCode));
fieldList.dynamic(QueryFieldListDynamic::No);

//Insert the field mapping between target table
 and query data sources
fieldMapping = new Map(Types::String, Types::Container);
fieldMapping.insert(fieldStr(PacktCustomerInvoices,
 InvoiceAccount),
 [qbdsCustInvoiceJour.uniqueId(),
 fieldStr(CustInvoiceJour, InvoiceAccount)]);
fieldMapping.insert(fieldStr(PacktCustomerInvoices,
 InvoiceId),
 [qbdsCustInvoiceJour.uniqueId(), fieldStr(CustInvoiceJour,
 InvoiceId)]);
fieldMapping.insert(fieldStr(PacktCustomerInvoices,
 InvoiceAmount),
 [qbdsCustInvoiceJour.uniqueId(),
 fieldStr(CustInvoiceJour, InvoiceAmount)]);
fieldMapping.insert(fieldStr(PacktCustomerInvoices,
 InvoiceQty),
  [qbdsCustInvoiceJour.uniqueId(),
fieldStr(CustInvoiceJour, Qty)]);
fieldMapping.insert(fieldStr(PacktCustomerInvoices,
 CurrencySymbol),
 [qbdsCustInvoiceJour.uniqueId(),
fieldStr(CustInvoiceJour, CurrencyCode)]);
```

```
        //Perform query insert recordset
        query::insert_recordset(customerInvoice, fieldMapping,
         query);
        //get date time on completion of process
        dateTimeNow         =   System.DateTime::get_UtcNow();

        //get time span by using subtract method of dateTime class
        timeSpan            =
         dateTimeNow.Subtract(utcDateTime2SystemDateTime
          (startDateTimeUTC));
        timeTaken           =   timeSpan.get_TotalMilliseconds();


        info(strFmt('It took %1 miliseconds to insert %2 records
         with
         a query::insert_recordset approach', timeTaken,i));
    }
}
```

7. The Infolog shows us `12490` records inserted in the table in `297` milliseconds using the `query::insert_RecordSet` approach:



How it works...

The `RecordInsertList` class acts as an array of the given type and holds the table buffer. It can insert multiple records in a single database trip. The `add()` method prepares the stack of records and the `insertDatabase()` method packs these records and sends them to the database in a single trip, where they are unpacked and inserted into the database table record by record. While creating an instance of `RecordInsertList`, we could skip certain system methods, such as insert by passing some parameters.

We have also used `insert_recordSet/delete_from/update_recordSet`, which are called **set based operations**, and insert all records in a database in a single call to the database. However, these operations are converted to record-by-record operations if insert/update/delete methods are overridden on application tables. To counter this, for example, to skip the insert method, we could specify `_tableBuffer.skipDataMethods(true)`.

There's more...

While working on a real-time project, you may need to deal with a number of records to perform CRUD methods. In such cases, it's a difficult and time consuming process to process each record individually. Let's look at how to handle such scenarios.

Using delete_from

We might need to delete records from a table specifying some clause in minimum time. This we could achieve using `delete_from`. Create a `deleteInvoicesDeleteFrom` class and add the following code:

```
class deleteInvoicesDeleteFrom
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    PacktCustomerInvoices   customerInvoice;
    int64                   timeTaken;
    System.DateTime         dateTimeNow;
    System.TimeSpan         timeSpan;
    utcdatetime             startDateTimeUTC;
    int64                   i;

    select count(RecId) from customerInvoice;
    i = customerInvoice.RecId;

    startDateTimeUTC = DateTimeUtil::utcNow();

    //deletes all record in one go.
    delete_from customerInvoice;
    dateTimeNow       =   System.DateTime::get_UtcNow();
    timeSpan          =
```

```
            dateTimeNow.Subtract(utcDateTime2SystemDateTime
             (startDateTimeUTC));
        timeTaken          =   timeSpan.get_TotalMilliseconds();

        info(strFmt('It took %1 miliseconds to delete %2 records with
          a delete_from approach', timeTaken,i));
    }
}
```

While running your code, you will get an output notification as follows:

Using update_recordSet for faster updates

Sometimes, we need to update some records in a table, which we might do using loops, which can be time consuming. Here, to update the records in a single application to database trip, we can use `update_recordSet`. Create an `updateInvoicesUpdateRecordSet` class and add the following code:

```
class updateInvoicesUpdateRecordSet
{
  /// <summary>
  /// Runs the class with the specified arguments.
  /// </summary>
  /// <param name = "_args">The specified arguments.</param>
  public static void main(Args _args)
  {
    PacktCustomerInvoices   customerInvoice;
    int64                   timeTaken;
    System.DateTime         dateTimeNow;
    System.TimeSpan         timeSpan;
    utcdatetime             startDateTimeUTC;
    int64                       i;

    select count(RecId) from customerInvoice
```

```
            where customerInvoice.InvoiceQty<100;;
        i = customerInvoice.RecId;
        customerInvoice.clear();
        startDateTimeUTC = DateTimeUtil::utcNow();
        ttsbegin;
        //updates all record in one go.
        update_recordset customerInvoice
            setting IsUpdated=1
            where customerInvoice.InvoiceQty<100;
        ttscommit;
        dateTimeNow         =   System.DateTime::get_UtcNow();
        timeSpan            =
          dateTimeNow.Subtract(utcDateTime2SystemDateTime
            (startDateTimeUTC));
        timeTaken           =   timeSpan.get_TotalMilliseconds();

        info(strFmt('It took %1 miliseconds to update %2 records
          with a update_recordset approach', timeTaken,i));
    }
}
```

Writing efficient SQL statements

In Dynamics 365 for Finance and Operations, SQL statements can often become performance bottlenecks. Therefore, it is very important to understand how Visual Studio handles database queries and to follow all the best practice recommendations in order to keep your system healthy and efficient.

In this recipe, we will discuss some of the best practices to use when writing database queries. For demonstration purposes, we will create a sample `find` method with different logic and queries and discuss each of them. The method will locate the `CustGroup` table record of the given customer account.

How to do it...

1. As methods are not allowed on extensions, to demonstrate this recipe we need to over layer a `CustGroup` table. Add a `CustGroup` table in your project, and create the following method:

```
/// <summary>
///
/// </summary>
```

```
/// <param name = "_custAccount"></param>
/// <param name = "_forupdate"></param>
/// <returns></returns>
 public static CustGroup PktfindByCustAccount(CustAccount
 _custAccount,
 boolean _forupdate = false)
{
  CustTable custTable;
  CustGroup custGroup;

  if (_custAccount)
  {
    select firstOnly CustGroup from custTable
    where custTable.AccountNum == _custAccount;
  }
  if (custTable.CustGroup)
  {
    if (_forupdate)
    {
        custGroup.selectForUpdate(_forupdate);
    }

    select firstOnly custGroup where
      custGroup.CustGroup == custTable.CustGroup;
  }
  return custGroup;
}
```

2. In the same table, create another method with the following code snippet:

```
/// <summary>
///
/// </summary>
/// <param name = "_custAccount"></param>
/// <param name = "_forupdate"></param>
/// <returns></returns>
public static CustGroup PktfindByCustAccount2(
  CustAccount _custAccount,
  boolean _forupdate = false)
{
 CustTable custTable;
 CustGroup custGroup;

 if (_custAccount)
 {
   if (_forupdate)
   {
      custGroup.selectForUpdate(_forupdate);
```

```
            }
            select firstOnly custGroup exists
            join custTable
            where custGroup.CustGroup == custTable.CustGroup
              && custTable.AccountNum == _custAccount;
        }
        return custGroup;
    }
```

How it works...

In this recipe, we have two different versions of the same method. Both methods are technically correct, but the second one is more efficient. Let's analyze each of them.

In the first method, we should pay attention to the following points:

- Verify that the `_custAccount` argument is not empty; this will avoid the running of an unnecessary database query.
- Use the `firstOnly` keyword in the first SQL statement to disable the effect of the read-ahead caching. If the `firstOnly` keyword is not present, the statement will retrieve a block of records, return the first one, and ignore the others. In this case, even though the customer account is a primary key and there is only one match, it is always recommended that you use the `firstOnly` keyword in the `find()` methods.
- In the same statement, specify the field list--the `CustGroup` field--we want to retrieve, instructing the system not to fetch any other fields that we are not planning to use. In general, this can also be done on the Application Object Tree query objects by setting the `Dynamic` property of the `Fields` node to `No` in the query data sources and adding only the required fields manually. This can also be done in forms by setting the `OnlyFetchActive` property to `Yes` in the form's data sources.
- Execute the `selectForUpdate()` method only if the `_forupdate` argument is set. Using the `if` statement is more efficient than calling the `selectForUpdate()` method with `false`.

The second method already uses all the discussed principles, plus an additional one:

- Both the SQL statements are combined into one using an `exists` join. One of the benefits is that only a single trip is made to the database. Another benefit is that no fields are retrieved from the `customer` table because of the `exists` join. This makes the statement even more efficient.

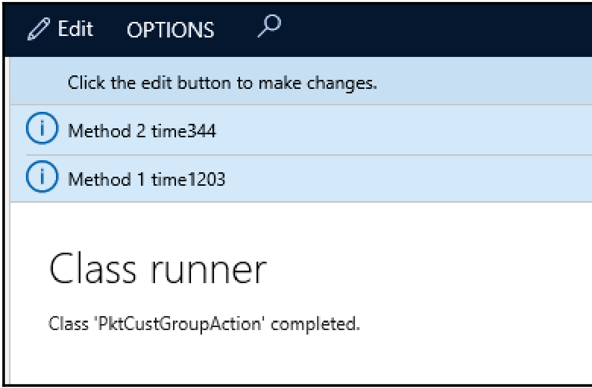
There's more...

To understand the preceding code in more depth, let's try to analyze the execution using one of the earlier recipes code. Carry out the following:

Add a new class in your project and write the following code:

```
class PktCustGroupAction
{
  public static void main(Args args)
  {
    CustGroup       custGroup;
    int64                   timeTaken;
    System.DateTime         dateTimeNow;
    System.TimeSpan         timeSpan;
    utcdatetime             startDateTimeUTC;

    startDateTimeUTC = DateTimeUtil::utcNow();
    CustGroup   = CustGroup::pktFindByCustAccount("10");
    // sleep(1000); // pause for 1000 milliseconds
    dateTimeNow       =   System.DateTime::get_UtcNow();
    timeSpan          =
      dateTimeNow.Subtract(utcDateTime2SystemDateTime
        (startDateTimeUTC));
    timeTaken         =   timeSpan.get_TotalMilliseconds();

    info(strFmt("Method 1 time%1", timeTaken ));

    startDateTimeUTC = DateTimeUtil::utcNow();
    CustGroup   = CustGroup::PktfindByCustAccount2("10");
    // sleep(1000); // pause for 1000 milliseconds
    dateTimeNow       =   System.DateTime::get_UtcNow();
    timeSpan          =
     dateTimeNow.Subtract(utcDateTime2SystemDateTime
       (startDateTimeUTC));
    timeTaken         =   timeSpan.get_TotalMilliseconds();
    info(strFmt("Method 2 time%1", timeTaken));
  }
}
```

Now, set your class and project as startup object/project. Save all your code and run the solution. You will get the following results:



Method 1, where we used different `if` statements to validate some conditions clearly shows that it took `1203` ms, while **Method 2**, where we used `if` statement wisely to optimize the code and execution time, took `344` ms. This is how you must use efficient code during development to reduce the time taken and speed up your execution time for better performance.

See also

There are many APIs that are deprecated in Dynamics 365 for Finance and Operations. You can have a look at the following link to check all the lists of deprecated APIs in Dynamics 365 for Finance and Operations. Along with others, `WINAPI` is no longer in use in the current version of Dynamics 365 for Finance and Operations:

- `https://docs.microsoft.com/en-us/dynamics365/operations/dev-itpro/migration-upgrade/deprecated-apis`

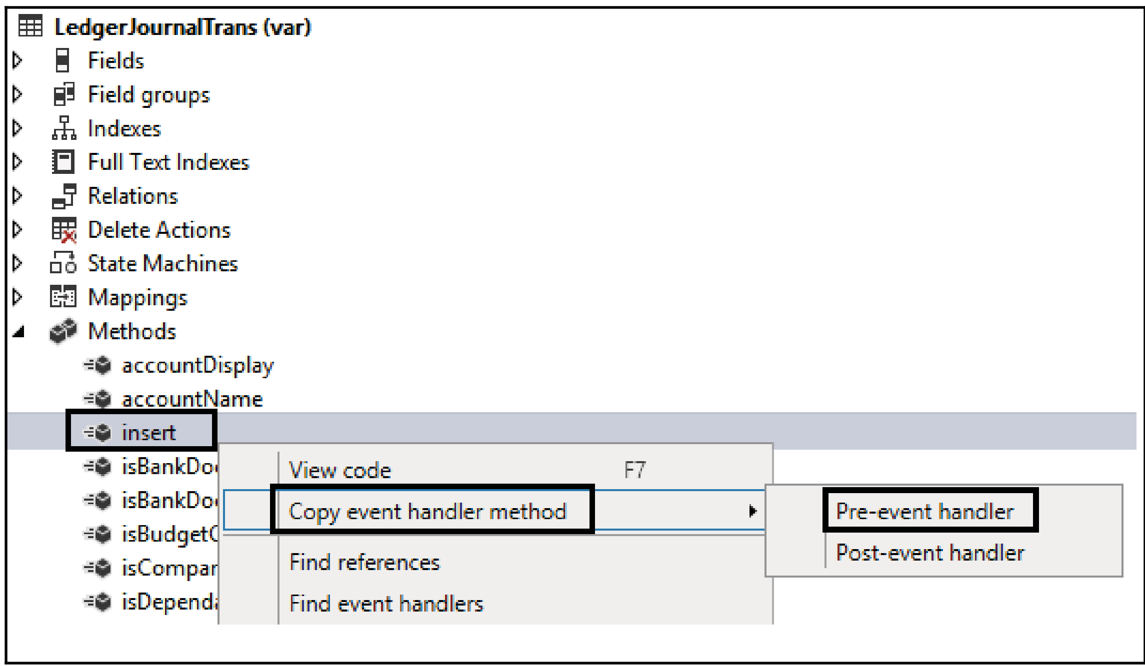
Using event handler

Use of event handler is always recommended, and it is the safest way to modify any existing functionality in Dynamics 365 for Finance and Operations. However, it may not fit with every requirement, but always try to use event handler in every possible place. You can use event handler on Classes, Forms, and Tables. On any method, whether it's on a Table, Class, or Form, you can write `pre` or `post` event handler, while on Tables and Forms you will also get standard Events under Event nodes such as `onInserting`, `onDeleted`, and so on.

To understand this concept better, let's take the example of General Journal. While entering new lines in Journal, we need to validate that the credit amount must not exceed 1000. To do this, carry on with the following recipe.

How to do it...

1. Go to the `LedgerJournalTrans` table and open it in designer.
2. Right-click on the `insert` method and select **Copy event handler method** | **Pre-event handler**:

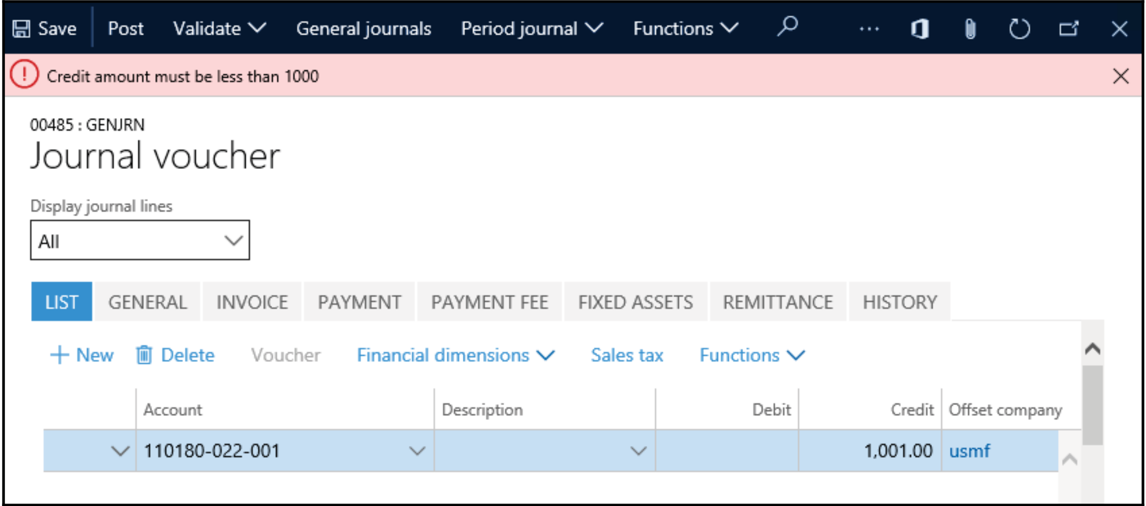


3. Create a new class in your project, give it the name `pktLedgerJournalTransEventHandler`, and paste event handler code here.
4. Now get the table buffer and write your logic as follows:

```
class PktLedgerJOurnalTransEventHandler
{
  /// <summary>
  /// Pre event handler method on Insert method to put
     validations
```

```
    /// </summary>
    /// <param name="args"></param>
    [PreHandlerFor(tableStr(LedgerJournalTrans),
       tableMethodStr(LedgerJournalTrans, insert))]
    public static void
    LedgerJournalTrans_Pre_insert(XppPrePostArgs
     args)
    {
      LedgerJournalTrans  ledgerJournalTrans = Args.getThis();
      if(ledgerJournalTrans.AmountCurCredit > 1000 )
      {
        Global::error("Credit amount must be less than 1000");
      }
    }
  }
```

5. Now, to test your code, go to **General Ledger** | **Journal Entries** | **General Journal** and create a new journal, click on lines, and create a new line with the credit amount `1001`. On saving, the system should throw the error **Credit Amount must be less than 1000**, as shown in the following screenshot:



6. You can add any logic or validation using Event handler, as per the requirement chosen from `pre or post` event handler.

How it works...

When you copy an event handler method, the system will copy its syntax and you can paste it in a new method in the event handler class. event handler uses `XppPrePostArgs` to pass the arguments and the same will be used to capture the table buffer using the `getThis()` method. Once you get the table buffer, you can retrieve any field or method to use in your logic.

On the basis of your selection criteria, the event handler method will execute. For example, if you choose post Event handler, it will execute just after that method execution is finished. Refer to the following code to understand post Event handler:

```
[PostHandlerFor(tableStr(CustGroup), tableMethodStr(CustGroup,
   delete))]
public static void CustGroup_Post_delete(XppPrePostArgs args)
{
}
similarly, if you choose Pre-event handler it will execute just
  before the execution of method.
[PreHandlerFor(tableStr(CustGroup), tableMethodStr(CustGroup,
 delete))]
public static void CustGroup_Pre_delete(XppPrePostArgs args)
{
}
```

There's more...

Post event handler is the same, but the only difference is that it will execute after the parent method. Apart from pre-and post-event handler, Dynamics 365 for Finance and Operations provides event handling on system events/methods such as `onDeleting`, `OnDelete`, `OnInsert`, and so on. These event handlers have different syntax than the Pre-Post event handler, but the concept is the same.

To use these event handlers, simply expand the **Events** node on **Tables and Forms**. Classes don't have Events. Expand the Events node and choose the required event, then right-click and select **Copy Event Handler Method** and paste it into your event handler class. You have to use the following syntax to get the table buffer:

```
[DataEventHandler(tableStr(LedgerJournalTrans),
   DataEventType::Inserted)]
public static void LedgerJournalTrans_onInserted(Common
  sender, DataEventArgs e)
{
   LedgerJournalTrans  ledgerJournalTrans = sender as
```

```
        ledgerJournalTrans;
       //Now use this table buffer in your logic
    }
```

Creating a Delegate method

The `Delegate` method is also very helpful to minimize overlaying, and you can use the `Delegate` method to communicate objects that exist in different packages/models and help to solve dependencies between models when migrating code. `Delegate` can be very useful when you need to use two objects that are not in the same package. Use the delegate concept by defining a contract between the delegate instance and the delegate handler. We have a new structure of Dynamic 365 for Operations--you will not be able to use an object outside of its own package. So, to use delegate in such situations a delegate declaration must have three things--a `Delegate` keyword, return type should be `void`, and it should be an empty method.

Let's understand it using the following recipe.

Getting ready...

To understand this recipe, let's consider a scenario. We have a requirement where we are creating an **Expense** journal through code and to identify such transactions we added a new field on the `LedgerJournalTrans` table as `PktNoLedgerPost`. Now, whenever users post this journal, it will go to the project journal instead of Ledger accounts.

To skip `Ledger` posting we need to check for this customized Boolean field and pass a `false` value to its `parmPostToGeneralLedger` method.

How to do it...

1. Add a `ProjPostCostJournal` class on your solution, and add a new delegate method with the following syntax:

```
/// <summary>
//New delegate method added by Deepak On May 11, 2017
/// </summary>
delegate void pktCheckNoLedger(RefRecId _Recid,
  EventHandlerResult _result)
{
```

```
}
Now we have to call this method in New method of
ProjPostCostJournal
 class. Add below code to call delegate method
Boolean noLedger;
EventHandlerResult  result = new EventHandlerResult();
if(ledgerJournalTrans.RecId)
{
  this.PktCheckNoLedger(ledgerJournalTrans.RecId, result);
  noLedger = result.result();
  this.parmPostToGeneralLedger(noLedger);
}
```

2. Add a new event handler class in your project and add a new method with the following code:

```
[SubscribesTo(classStr(ProjPostCostJournal),
 delegateStr(ProjPostCostJournal, pktCheckNoLedger))]
 public static void ProjPostCostJournal_PktCheckNoLedger
   (RefRecId _Recid, EventHandlerResult _result)
 {
    LedgerJournalTrans ledgerJournalTrans =
      LedgerJournalTrans::findRecId(_Recid, false);
    if(ledgerJournalTrans.PktNoLedgerPost)
    {
        _result.result(false);
    }
    else
    {
        _result.result(true);
    }
 }
```

3. Now try to post one **Expense** journal with this scenario and you will see there are no General Journal transactions; all vouchers will post only in Project transactions.

How it works...

Delegate methods serve as a contract between the delegate instance and the delegate handler, without any code/logic itself. The `SubscribesTo` keyword in an event handler method creates a static delegate handler. `SubscribesTo` requires a class name and delegate method name.

We created a new delegate method and called this method in between the new method. When a pointer calls this method, it will call event handler code and we will get the required buffer.

There's more...

`EventHandlerResult` is used to get any return value from event handler to use in your code. However, it's not mandatory and you can create as many parameters as required. So, try to add more parameters to explore and play with the `Delegate` method.

See also

Please have a look at the `DimensionHierarchyDelegates` class and how it works. This class has many delegate methods and they are invoked from the `DimensionFocus` form. Try to debug these two objects to better understand the various uses of delegate methods.

Index

A

Application Object Tree (AOT) 189, 401
automatic lookup
  creating 180, 183
automatic transaction text
  modifying 250, 254
Azure Active Directory (AAD) 378
Azure Blob 342
Azure portal
  URL 379
Azure SQL
  URL, for connecting to Power BI 367
Azure
  references 387

B

Boolean value 244
browse
  building, for folder lookups 213

C

caching
  display method 420
code execution time
  calculating 424
color picker lookup
  creating 219, 222
color selection dialog boxes
  using 219
confidential client 378
CRUD (Create, Read, Update, and delete) 401
currency converter
  reference link 406
custom filter control
  creating 132, 137
custom instant search filter
  creating 138, 141
custom Lookup
  creating 340, 341
custom options
  displaying, in another way 200, 206
  list, displaying 197, 200
custom service
  consuming, in JSON 392
  consuming, in SOAP 397, 401
  creating 388, 392

D

data consistency checks
  enhancing 53, 58
data contract 77
data entities
  about 274, 280, 397
  building, with multiple data sources 283
  document 282
  master 281
  parameters 281
  reference 281
  transaction 282
data management 273
data migration 308
data packages
  about 292, 302
  reference link 302
data
  consuming, in Excel 356
  importing 308, 314
date effectiveness feature
  using 58, 62
Delegate method
  creating 445, 447
delete operation
  enhancing 427

somewhere clause, using 435
dialogs
creating, RunBase framework used 64, 70
creating, SysOperation framework used 76, 84
event, handling 70, 76
direct SQL statement
executing 47, 51, 53
display method
caching 420
document handling note
adding 28, 32
document management 342, 344
dynamic form
building 87, 92
Dynamics 365
about 223
APIs 245

E

EDT (Extended Data Type) 180
electronic payment format
creating 266, 272
elements 9
Event handler
using 441
Excel Data Connector app
configuring 328, 332
using 328, 332
Excel
data, consuming 356
integrating, with Power BI 363, 366
export API 336, 339
export process 303
extensions
using 416, 419
external web services
consuming 406

F

folder browsing lookups 213
folder lookup
browse, building 213
form pattern
list 117, 120
selecting 116
form splitter
adding 93, 96
form
about 64
creating 121, 126
using, to build lookups 187, 192

G

general journal
creating 230, 235, 238
posting 238, 240

H

hardcoded options 197

I

image
adding, to records 170, 172
displaying, as part of form 173, 174
stored image, saving as file 175
import process 308
insert operation
enhancing 427
interactive dashboards
developing 366, 374

J

JSON (JavaScript Object Notation)
custom services, consuming 392

L

labels 397
last form values
storing 101, 105
ledger voucher processing 245
ledger voucher
creating 245, 250
posting 245, 250
Lifecycle Services (LCS) 292
Line records 263
lookups
about 179
building, based on record description 207, 213
building, form used 187, 192

creating, dynamically 183, 187

M

macro
using, in SQL statement 44, 46
Microsoft.IdentityModel.Clients.ActiveDirectory library
reference link 378
modal form
creating 96
model
creating 9
multiple data sources
data entity, building 283
multiple forms
modifying, dynamically 98, 101
multiple records
processing 164, 167
MVC (Model-View-Controller) 76

N

native client app
aadResource 387
authenticating 378, 386
client App Id 387
PlatformParameters 387
redirect URI 387
native client application 378
normal table
using, as temporary table 32
number sequence handler
using 128, 131
number sequence
creating 14, 21

O

OData endpoint
reference link 361
OData services
consuming 401, 406
reference link 406
operations
reference link 441

P

package
creating 9
Power BI
about 346
configuring 347, 355
Excel, integrating 363, 366
reference link 347
URL 363
URL, for downloading 355
visuals, embedding 374, 376
Power Query
URL, for downloading 366
Power view option
URL, for enabling 362
primary key
renaming 22, 28
project journal
processing 241, 244
project
creating 9
Purchase Order Header record 257
purchase order
creating 254, 257
posting 257, 260

Q

query object
building 39, 42
OR operator, using 43

R

records
coloring 168, 170
copying 34, 39
image, adding 170, 172
REST (Representational State Transfer) 401
RunBase framework
used, for creating dialogs 64, 70

S

Sales Order Header 263
sales order
creating 261

posting 264, 265
segmented entry control
using 224, 226, 229
selected/available list
building 141, 147, 150
set based operation 435
SharePoint online 342
SOAP (Simple Object Access Protocol)
about 392
custom services, consuming 397, 401
Soap Utility 397
splitters 93
SQL statement
macro, using 44, 46
writing 437, 441
SysOperation framework
asynchronous 77
reliable asynchronous 77
scheduled batch 77
synchronous 76
used, for creating dialogs 76, 84

T

tree control
using 105, 112
tree lookup
building 193, 197
troubleshooting 314, 326

U

unique record identification value 207
update operation
enhancing 427
update_recordSet, using 436

V

View details link
adding 112, 115

W

wizard
creating 151, 155, 163
Workbook Designer
using 333, 336

X

X++ code 183