PROGRAMMING IN AL

FOR BEGINNERS

VERSION 2

TABLE OF CONTENT

INTRODUCTION

Microsoft Dynamics 356 Business Central is an ERP system that can be easily customized with AL language extensions. This workbook has been prepared to guide you through the basics of development.

I hope that after reading it and doing all the tasks you will get familiar with the basics of developing in the AL Language. You should know that this is only the start. There are much more functionalities in the programming language but here you will see the essentials which will give you the fundaments to build more advanced extensions.

How should you work with it? First, it is not a book. I would like to encourage you to print it (if you already do not have a hard copy) and write in it. Make some notes and write down the things which you would like to explore more. That is why I am calling it WORKBOOK.

In the next pages, you will see some code that you need to write. I would like to ask you something. **Do not copy it!** Try to write it by yourself. Use this which you can find here as an example and reference. This way you will understand more and get familiar with how to do it later by yourself.

I hope you will like the structure and the content. If so, please share some thoughts on social media such as LinkedIn, and Twitter or comment on the blog **MyNAVBlog.com**.

The material here is fully free of charge. If you like you can use it in your organization or prepare the workshops for your colleagues. Please only do not remove copyrights from the materials.

Krzysztof Bialowas

www.MyNAVBlog.com

PROJECT REPOSITORY

Whole code used in this workbook and also the newest version of this workbook you can find on the GitHub page: **https://github.com/mynavblog/ALForBeginners**

FEW WORDS ABOUT THIS VERSION

After I prepared the first version of this workbook a lot has changed. First of all, the current version of Dynamics 365 Business Central is 20.3 and many things have been added to the language since 2019. I decided to update the workbook since a lot of people start using it and had comments (thank you Steve for pushing me for this update).

I updated the way how to work with docker and focus less on non-important things. Also added more exercises that were inspired by people who were new to Business Central (Radek thank you for all your comments).

The general idea about is workbook did not change. Still, it is for people which are starting to learn AL development.

From a technical perspective, I changed the tools used to prepare the workbook – the reason now more people can contribute to it.

July 2022

CHAPTER 1

PREPARING THE DEVELOPMENT ENVIRONMENT

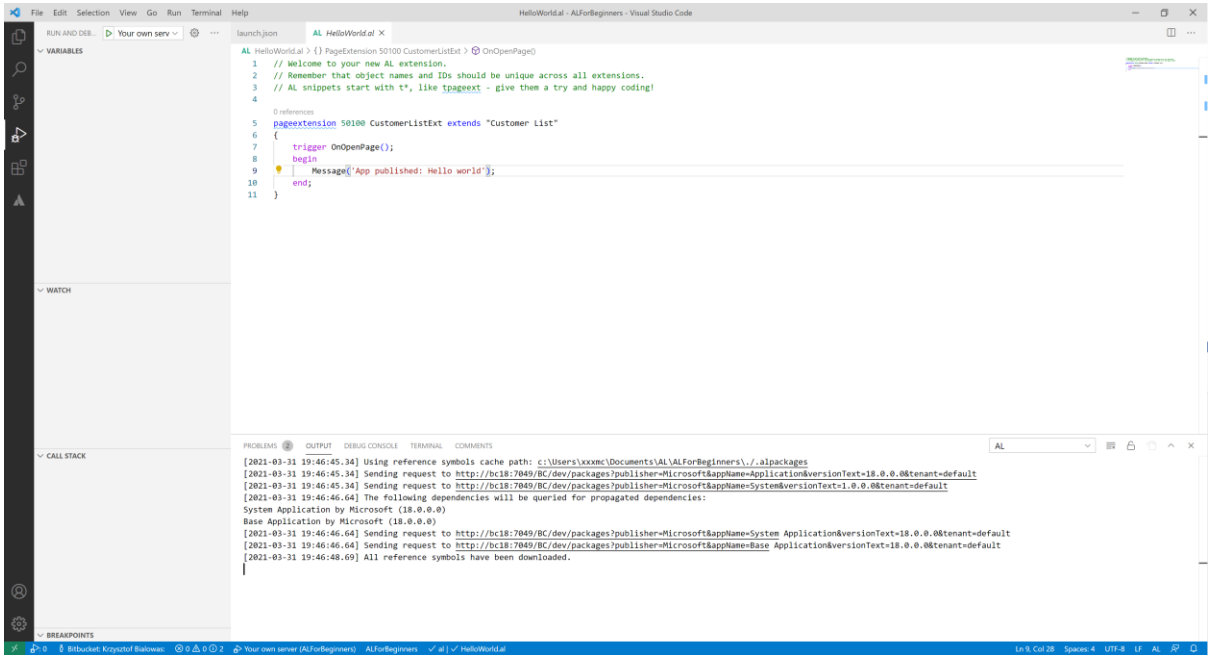
OBJECTIVES

To start building extensions for Microsoft Dynamics 365 Business Central you need to prepare the development environment first. Only Visual Studio Code is mandatory for the development however in this chapter you will get knowledge on how to:

- ✓ Install Visual Studio Code with extensions for AL language
- ✓ Setup development database using Docker Container
- ✓ Setup development database directly on Online Sandbox

VISUAL STUDIO CODE

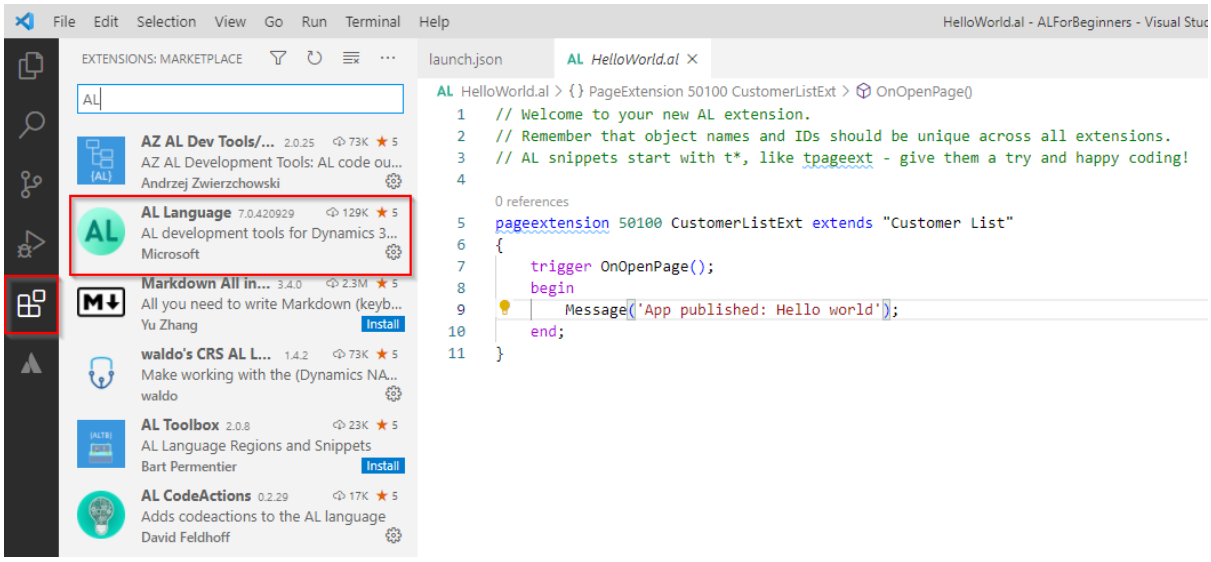
Visual Studio Code (VSCode) is the code editor for the Dynamics 365 Business Central development. It allows us to write and debug the code in the AL language in which the extensions are written. In the next chapters, you will get familiar with more functions and how to use them.



INSTALLATION

To install Visual Studio Code go to page https://code.visualstudio.com and download the current version for your platform.

After installation open Visual Studio Code and go to **Extensions** management and find the **AL Language**. Install it.



HINTS

Visual Studio Code gives us the possibility to extend and configure it. In the marketplace (Extension Management) you can find many useful addons that can be installed to help you work with the AL language.

If the version on the screen above is different than the one on your computer do not worry. It is because Microsoft updates frequently this extension.

*I recommend checking **AL Extension Pack by Waldo**. It contains the most important extensions for AL development. Do not install them all at once but check which are the best for you*

In this workbook, all screens are done with a white background but to work I recommend you change the color theme to dark (if you did not do it already).

DOCKER AND LOCAL DEVELOPMENT ENVIRONMENT

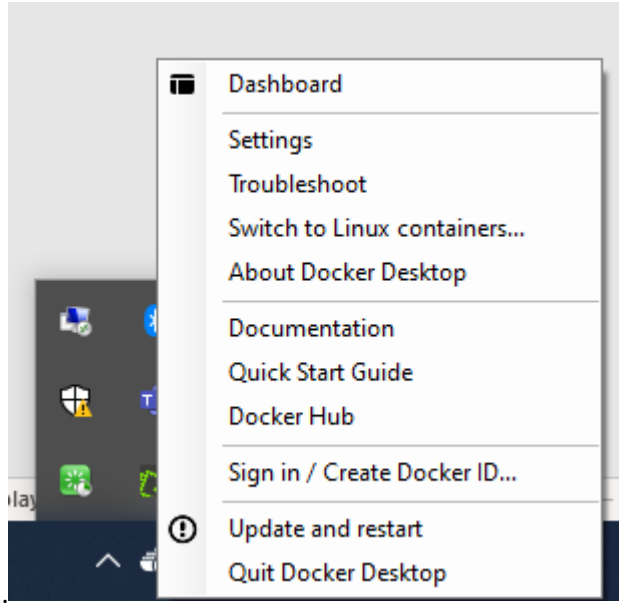
Docker allows you to create local development environments. Each environment is separated and packed in something called a container. Thanks to it you can have more than one installation of Dynamics 365 Business Central on your computer.

Creating a container is very easy and can be done using PowerShell Script and a special gallery.

INSTALLATION

To install Docker, go to the page https://www.docker.com. And click the option Get Started. You will need to create a free account to download Docker.

After installation, please make sure that your Docker is running in Windows container Mode. To check that, in the tray bar, click right on the Docker icon and check if you have the option Switch to Linux Containers. If yes, then it means that your Docker works as Windows containers. You can also mark this option during the installation



Creating the first Business Central container

To create a Business Central Docker container, you can use the PowerShell script. Install **BCContainerHelper** module. You can find out more about the library on the page: https://github.com/microsoft/navcontainerhelper.

The below script allows you to install the **BCContainerHelper** module.

Script

```
Install-Module BcContainerHelper
Import-Module BcContainerHelper
```

Hint

Sometimes on your machine, you cannot run the PowerShell Scripts. To change the settings, you can run the script presented below.

SCRIPT

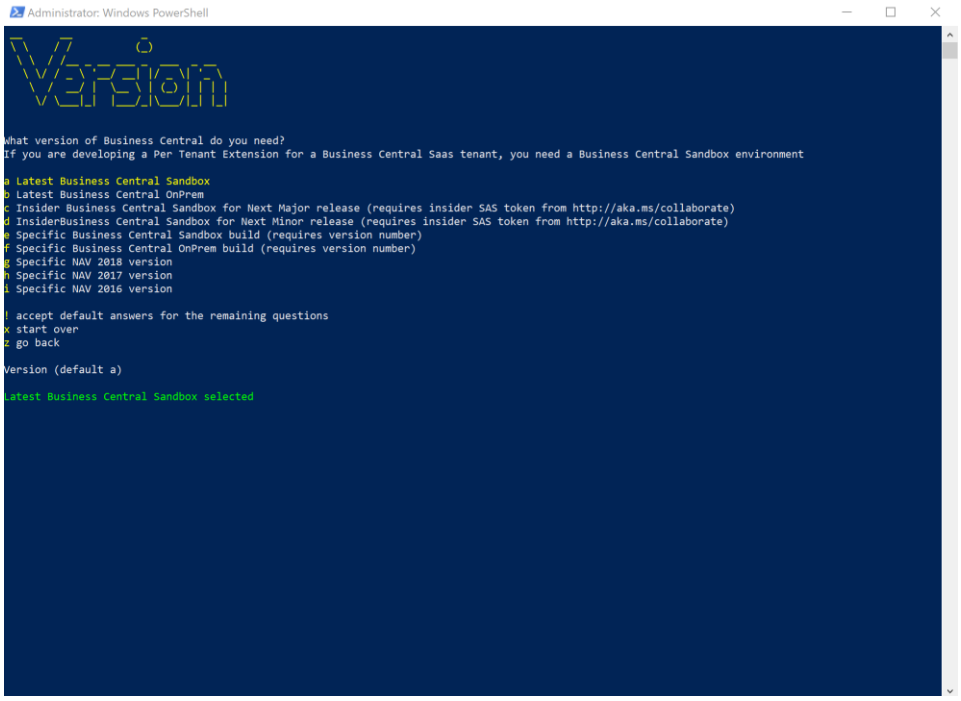
```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Scope CurrentUse
```

After installing BCContainerHelper you can run the below script to create a docker container with Business Central. After installing BCContainerHelper you can run the below script to create a docker container with Business Central. The special step-by-step guide will walk you thru all possible options. The special step-by-step guide will walk you thru all possible options.

My recommendation is to create your first container:

SCRIPT

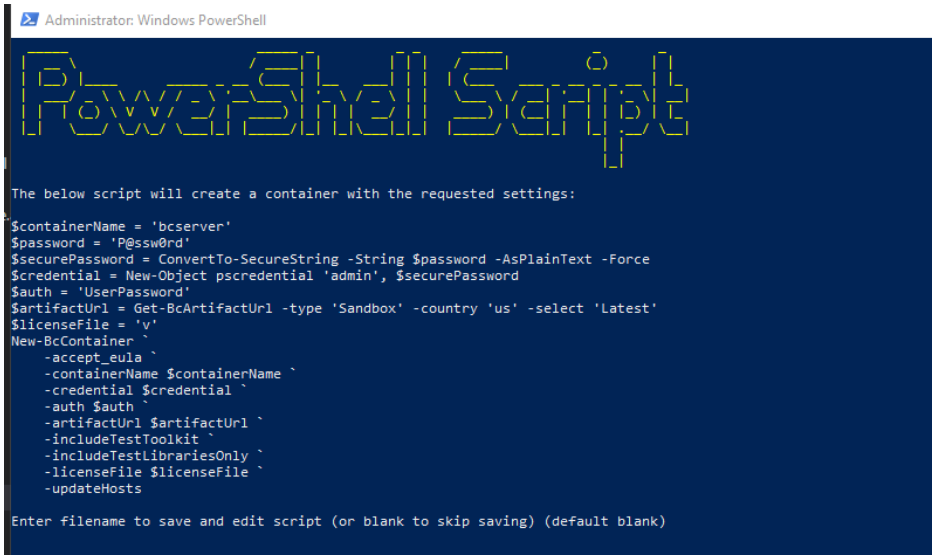
```
New-BcContainerWizard
```



My recommendation is to create your first container with the below parameters (the order of parameters represents steps in the wizard, note also that most of the options are default options for the wizard):

- ✓ Choose Local Container
- ✓ Use predefined password
- ✓ Choose your container name (since later you will have more than one)
- ✓ Latest Business Central Sandbox
- ✓ If you do not have any country preferences – choose us
- ✓ Install Test Framework and Test Libraries (it will be needed to create automated tests and finish the workbook)
- ✓ Performance Toolkit is not required at this moment
- ✓ The premium plan is not mandatory for exercises in the workbook
- ✓ Test users are not required
- ✓ AL BaseApp development is not required
- ✓ Use language extension provided with a container
- ✓ A custom license is not required for exercises in the workbook
- ✓ Use Cronus demo database (it will have some default data inside)
- ✓ Use multitenant
- ✓ Use default DNS settings

- ✓ Do not use DNS
- ✓ Allow the ContainerHelper to decide which isolation mode to use
- ✓ Use the default option for memory limit
- ✓ Skip saving image
- ✓ You can save the script to not start from scratch one more time otherwise execute the script



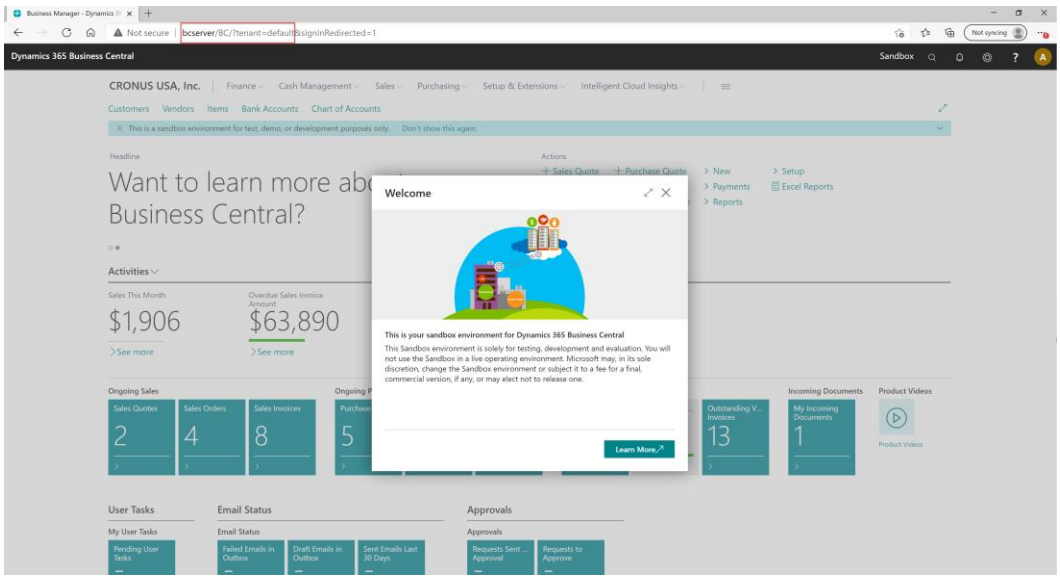
SCRIPT

```
New-BcContainerWizard
```

After the script will be executed the container will be running automatically. You can connect to it from a web browser using the address **Error! Hyperlink reference not valid.** container name>/BC/?tenant=default (so if you used the default name the address would be: **http://bcserver/BC/?tenant=default** . Then information you can find also in the output of the PowerShell script.

```
Enabling rewrite rule: Hostname (without port) to tenant
Dismounting Tenant
Mounting Tenant
Mounting Database for default on server localhost\SQLEXPRESS with AllowAppDatabaseWrite = False
Sync'ing Tenant
Tenant is Operational
Creating http download site
Setting SA Password and enabling SA
Creating admin as SQL User and add to sysadmin
Creating SUPER user
Container IP Address: 172.27.23.199
Container Hostname  : bcserver
Container Dns Name  : bcserver
Web Client          : http://bcserver/BC/?tenant=default
Dev. Server         : http://bcserver
Dev. ServerInstance : BC
Dev. Server Tenant  : default
Setting bcserver to 172.27.23.199 in host hosts file
Setting bcserver-default to 172.27.23.199 in host hosts file

Files:
http://bcserver:8080/ALLanguage.vsix

WARNING: You are running a container which is 131 days old.
Microsoft recommends that you always run the latest version of our containers.

Container Total Physical Memory is 8.5Gb
Container Free Physical Memory is 6.0Gb
```

The other chapter will explain how to connect to the container to start development.



HINTS

Running the script the first time can take time to download the artifact with the Business Central version.

If you would need to see the output of the script one more time you can run in PowerShell: ***docker logs <name of your container>*** *(for example* ***docker logs bcserver****).*

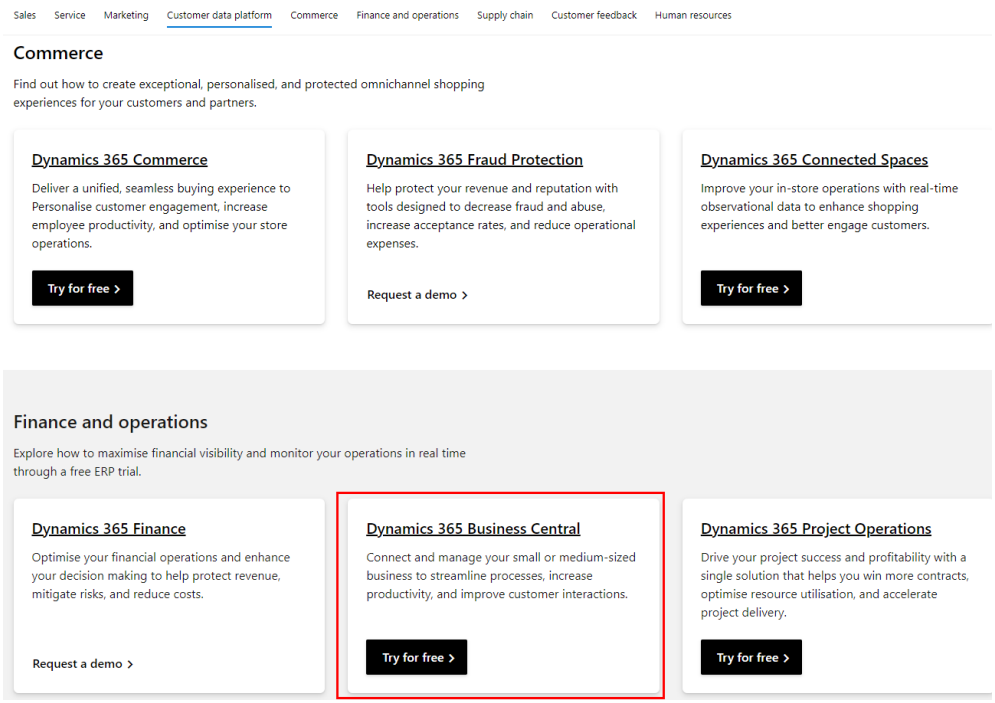
You can learn more about BCContainerHelper from the blog https://freddysblog.com/

ONLINE SANDBOX ENVIRONMENT

The Sandbox is the alternative to work with Docker containers. It is not available in all countries therefore you may not be able to create it if Business Central is not supported in your region.

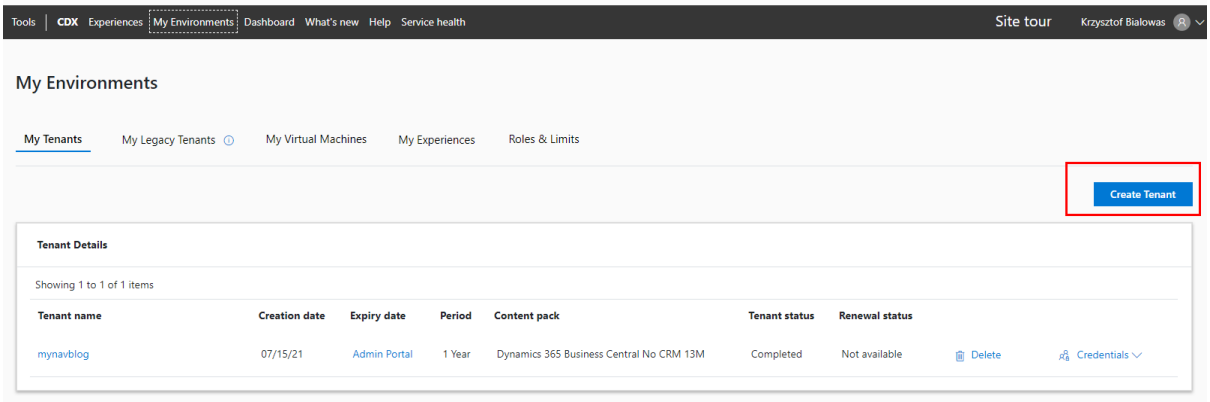
Go to page https://dynamics.microsoft.com/en-gb/dynamics-365-free-trial/ and find Business Central. Fill in your work e-mail address and phone.

Follow the steps on the screen to create a sandbox. The other chapter will explain how to connect to the online sandbox.



HINTS

Instead of using the link above you can create the environment with Business Central on cdx.transform.microsoft.com. The environment will be valid for either 3 months or 1 year depending on the option that you chose. The user and password will be then provided to you.



VERSION CONTROL AND REPOSITORY

When doing development for Business Central there is no common database where the code is stored. Each developer works on their own machine. The code later needs to be merged to the common repository which is used to build the test or production environment.

That is why it is needed to track the changes with the version control system. The most popular is Git. It allows skipping documenting the changes directly in the code which means the code is cleaner. Additionally, it controls when and by whom the changes were done.

You can install Git from the website https://git-scm.com/ https://git-scm.com and later use it in Visual Studio Code. In this workbook, I will not focus on using Git but to develop Business Central extensions it is good to know basic functions such as:

- ✓ **Clone** – allows cloning remote repository to a local folder
- ✓ **Pull** – allows to fetch and merge the changes from the remote repository
- ✓ **Commit** – allows committing all changes in the files with a meaningful description
- ✓ **Push** – allows pushing local and committed changes to the remote repository

You can store your code in source code management such as Azure DevOps, GitHub, or Bitbucket. Each of them allows the creation free repository.

HINTS

You can use Visual Studio Code to run Git functions either using a terminal (if you want to write them manually) or from the source control menu on the left side.

Even if you are working alone on the project it is good to use a remote repository to store the code.

There are many Git tutorials on the internet. I recommend checking https://www.atlassian.com/git/tutorials.

Many companies which implement Dynamics 365 Business Central use Azure DevOps to store the code, tasks, and building and deploy the solutions. You can create an account on the website https://dev.azure.com. However, more and more repositories for Business Central extensions are created also on GitHub. Please check https://github.com/microsoft/AL-Go/#readme

CHAPTER SUMMARY

- ✓ In this chapter, we did not do any development, but we prepared the environment.
- ✓ You can choose if you want to publish your extensions in the online sandbox or if you want to use a Docker container for that.
- ✓ You also will be able to manage your code on the remote repository.
- ✓ After the chapter, you have got everything installed for extension development for Business Central.

CHAPTER 2

FIRST EXTENSION FOR BUSINESS CENTRAL

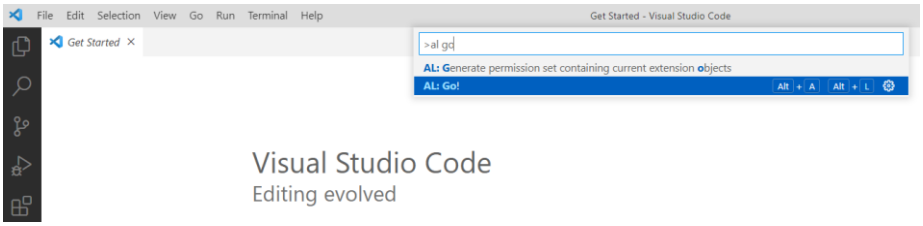
OBJECTIVES

After preparing the environment it is time to do the first development for Business Central. In almost every programming language the first program is to show "Hello World" on the screen. In this chapter, you will do the same. The objectives are:

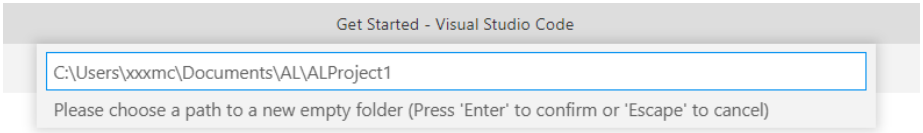
- ✓ Create a new AL project in Visual Studio Code
- ✓ Connect to Business Central
- ✓ Get familiar with common files in the project
- ✓ Publish the Hello World extension to the development environment

FIRST PROJECT

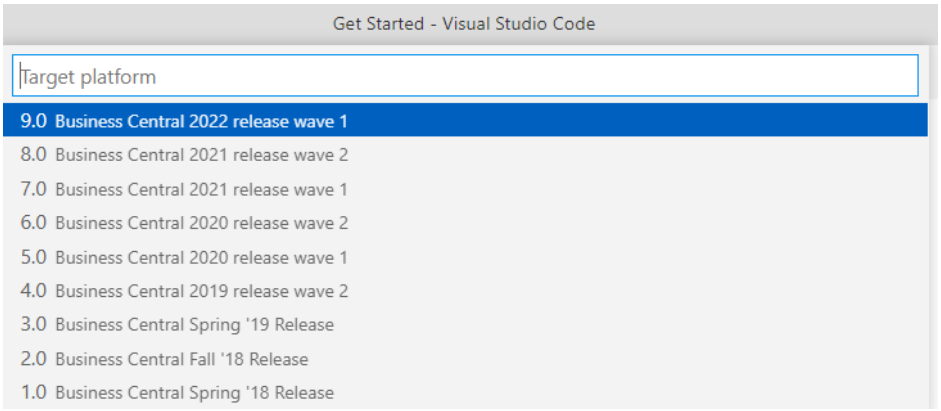
To create a new project for Business Central open Visual Studio Code and then open Command Palette. The key shortcut for that is **Ctrl+Shift+P** (or **F1**). Then write **AL: Go!**.



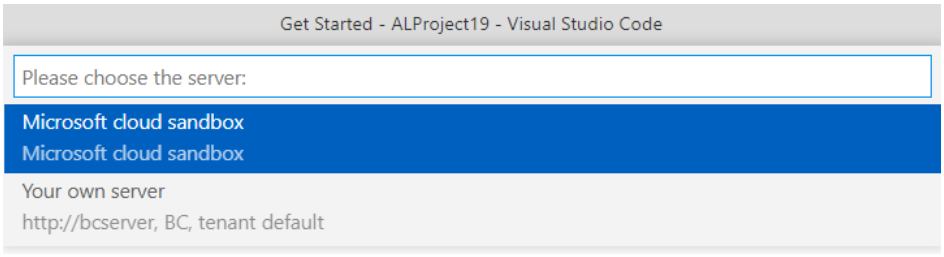
You can also use the key shortcut **Alt+A**, **Alt+L**.



After specifying where the project should be stored, you will need to choose for which platform you are doing development. In this workbook, examples will be developed for the newest available platform.

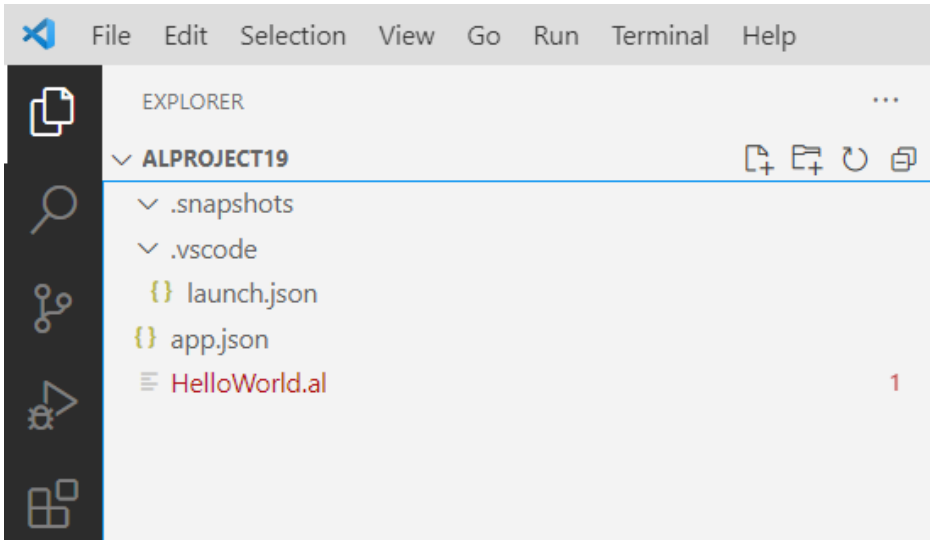


The next step is to decide if your development environment will be an online sandbox or your own Docker container. If you would choose the online sandbox you will be prompted to log in to it. At this moment click escape – how to connect to any of the two below will be described in part related to the **lauch.json** file.



After that, your first application is created. For now, it will contain primary three files:

- ✓ launch.json – in folder .vscode
- ✓ app.json
- ✓ HelloWorld.al



HINTS

*At this moment your **HelloWorld.al** file can show errors. Do not worry about them at this stage. The same situation can be with an **app.json** file.*

*You may also see empty folder **.snapshots**. This folder will not be described at this stage. Keep it in the solution.*

LAUNCH.JSON – HOW TO CONNECT TO BUSINESS CENTRAL

This file describes where the development environment is placed. It is possible to have multiple configurations at the same time.

HINTS

*If your file does not have any configuration yet. Click **Ctrl+Space** to see available options. Then you can choose one of two options: AL: Publish: Microsoft cloud sandbox or AL: Publish: Your own server.*

Configuration for the Online Sandbox

If you are using an online sandbox, then your **launch.json** file should look similar to the below.

.vscode > {} launch.json > ...

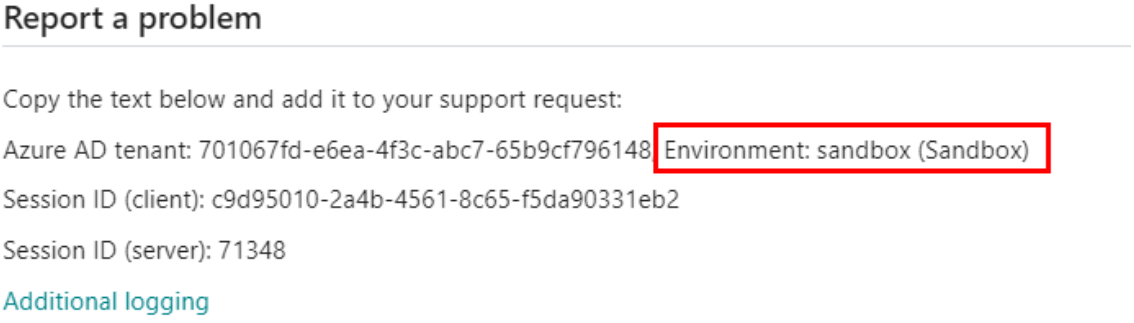
```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Microsoft cloud sandbox",
            "request": "launch",
            "type": "al",
            "environmentType": "Sandbox",
            "environmentName": "sandbox",
            "startupObjectId": 22,
            "startupObjectType": "Page",
            "breakOnError": true,
            "launchBrowser": true,
            "enableLongRunningSqlStatements": true,
            "enableSqlInformationDebugger": true
        }
    ]
}
```

Two of the most important parameters at this moment in this configuration are:

- ✓ **environmentType** – specifies what is the environment type to which you will login. It should always be **Sandbox.**
- ✓ **environmentName** – specifies what is the environment name where the development will be published. It can be different from the default one.

Hints

You can find what is the name of your environment when opening in Business Central Help & Support



Configuration for the Docker Sandbox

If you are using a Docker sandbox, then your **launch.json** file should look similar to the below.

.vscode > {} launch.json > ...

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Publish: Your own server",
            "type": "al",
            "request": "launch",
            "environmentType": "OnPrem",
            "server": "http://bcserver",
            "serverInstance": "BC",
            "authentication": "UserPassword",
            "startupObjectId": 22,
            "breakOnError": true,
            "breakOnRecordWrite": false,
            "launchBrowser": true,
            "enableSqlInformationDebugger": true,
            "enableLongRunningSqlStatements": true,
            "longRunningSqlStatementsThreshold": 500,
            "numberOfSqlStatements": 10,
            "tenant": "default"
        }
    ]
}
```

Two of the most important parameters at this moment in this configuration are:

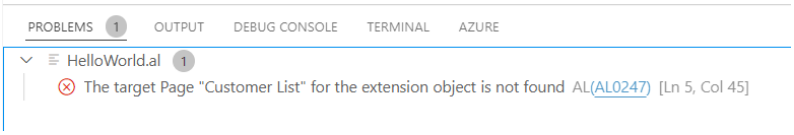
- ✓ **server** – specifies what is the http address of the docker instance. It should be the same as the name of the Docker container which you created. For example, if your Docker name is **bcserver** then the value in the property would be **http://bcserver**. Note that it would be different if you used SSL in your container (then it would be **https://bcsever**).

Hints

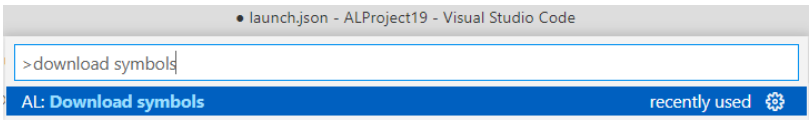
One of the parameters that you can add to configuration, regardless if this is an online sandbox or Docker container is "schemaUpdateMode": "ForceSync". This parameter allows in the later stage of this workbook to avoid errors when doing changes to the object names or numbers.

Download Symbols

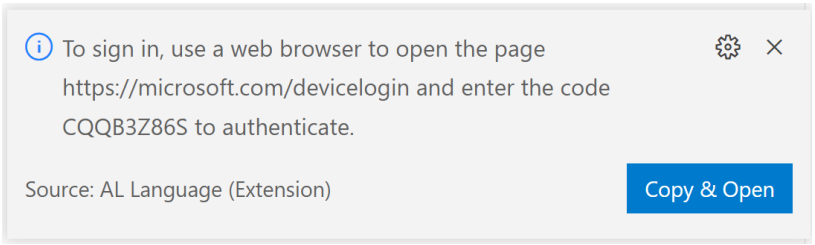
Probably, when you created a new project and opened **HelloWorld.al** file, you have at least one error. It says that the target page "Customer List" for the extension is not found. This is because you do not download symbols for standard Business Central objects.



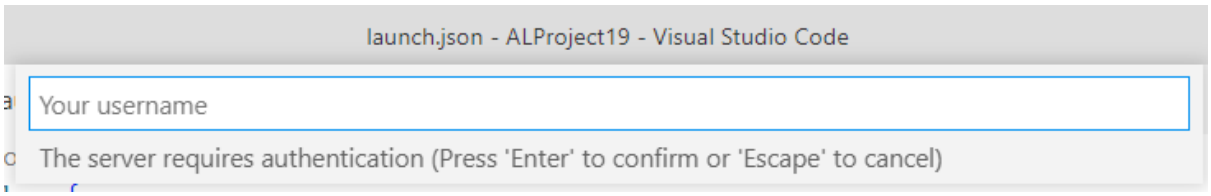
To download symbols, you need to open Command Pallet (F1) and run function AL: Download symbols.



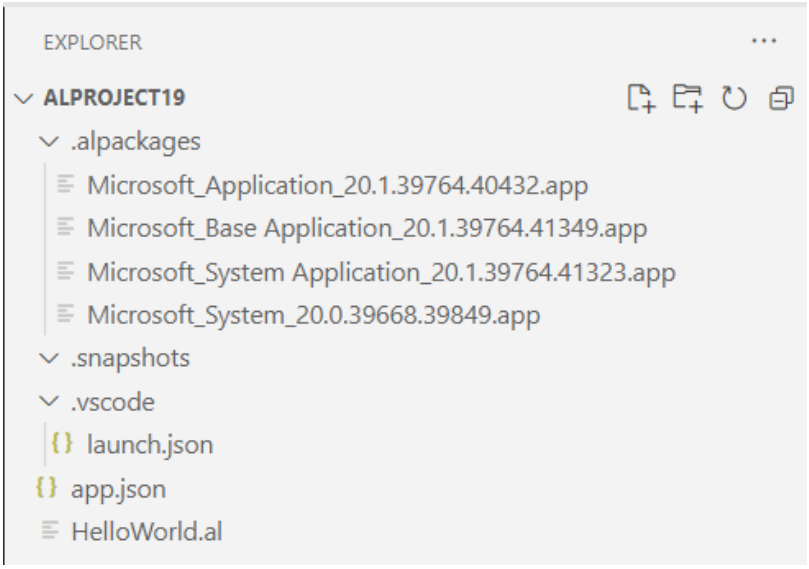
If you run the function for the first time, you will need to log in to your environment. This step will look different depending on the environment. If you are using an online sandbox, then you will see the below message in the right bottom corner. Follow the steps after clicking **Copy & Open**.



If you are using a Docker environment then on top of Visual Studio Code you will see the below window to specify the username and after confirmation, you will need to specify the password for the user. Use the same as you used when creating the container.



Symbols can be found in the folder **.alpackages** (generated automatically). If everything would be correct, you should find in that new folder four files with needed symbols. And the errors in the files will be resolved.



HINTS

If you do not have symbols in your folder most likely there is some issue with your launch.json file or with the login and password that you provided.

APP.JSON – INFORMATION ABOUT YOUR EXTENSION

This file provides the most important information about the extension which you develop. You can find here a name and a description of your extension, and information about a publisher. Also, you can specify the web addresses for a help site, end-user license agreements (EULA), and more.

app.json > ...

```
{
  "id": "4d35d9fd-22c9-452b-89c5-cf74cd47edda",
  "name": "ALProject19",
  "publisher": "Default publisher",
  "version": "1.0.0.0",
  "brief": "",
  "description": "",
  "privacyStatement": "",
  "EULA": "",
  "help": "",
  "url": "",
  "logo": "",
  "dependencies": [],
  "screenshots": [],
  "platform": "1.0.0.0",
  "application": "20.0.0.0",
  "idRanges": [
    {
      "from": 50100,
      "to": 50149
    }
  ],
  "resourceExposurePolicy": {
    "allowDebugging": true,
    "allowDownloadingSource": false,
    "includeSourceInSymbolFile": false
  },
  "runtime": "9.0"
}
```

At this moment it is not mandatory to change any property but during executing the exercises some of the properties will be populated. One of the properties that you can change at this stage is:

- ✓ **publisher** – specifies the company name that is developing the extension (app).

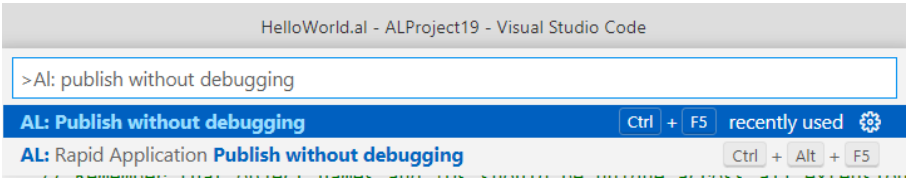
HelloWorld.al – First code in the extension

This file is your first code. It simply shows the message when you open the **Customers List**. In the next chapters of this workbook we will remove this file but for now, let's keep it. From this file, you can see that the extension of the files is ***.al**.

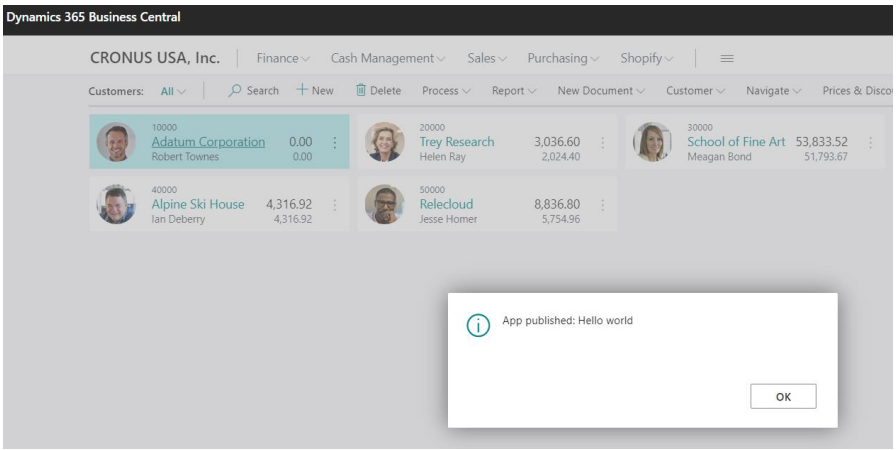
```
HelloWorld.al > { } PageExtension 50100 CustomerListExt
1   // Welcome to your new AL extension.
2   // Remember that object names and IDs should be unique across all extensions.
3   // AL snippets start with t*, like tpageext - give them a try and happy coding!
4
    0 references
5   pageextension 50100 CustomerListExt extends "Customer List"
6   {
7       trigger OnOpenPage();
8       begin
9           Message('App published: Hello world');
10      end;
11  }
```

Publish Code to Business Central

After you connect to Business Central it is time to publish the first extension. To publish the extension open Command Pallet (F1) and run function **AL: Publish without debugging**. You can also use the shortcut **Ctrl+F5.**



The web page with your Business Central will be opened and you should see the message whenever you open the **Customer List**.



CHAPTER SUMMARY

- ✓ In this chapter, you developed the first extension for Business Central in the AL language. Congratulations!
- ✓ Now you also know what you can find in the app.json and lanuch.json
- ✓ You also learned how to connect to an online sandbox or Docker container to start development

CHAPTER 3

EXTENSION OVERVIEW, PROJECT SETTINGS, AND STRUCTURE

OBJECTIVES

In this chapter, you will find out the best practices needed to start developing the extensions in the AL language. You will also get information on what you will be working on in the next chapters. The objectives are:

- ✓ Get familiar with the extension overview
- ✓ Enable the Code Analysis Tool
- ✓ Get familiar with the affixes
- ✓ Get familiar with object naming best practices

Bonus registration extension

Your company, for one of your customers, just decided to start a new project - **Bonus Registration** extension for Business Central, and your management decided that you will be working on it. Isn't that great?

What the extension should do?

Here are some bullet points from your system architect:

- ✓ First, the user should be able to create the Bonus Card. It should be possible to tell for which Bill-to Customer the bonus is created and what starting and ending date is.
- ✓ The bonus should be calculated only for items. Users should be able to decide if the bonus is for a particular item or all items.
- ✓ It must be possible to tell what the bonus percent per item is.
- ✓ There should be two statuses for the Bonus Card: *Open* when the bonus is not calculated, *Released* when the bonus is calculated and the Bonus Card is not editable.
- ✓ The users should be able to see for which posted sales lines bonus was granted. On the Bonus Card, users should see the total amount of the bonus.
- ✓ Users should be able to print a simple report for the bonus.

Quite a lot of points but in the next chapters you will find out how to develop such an extension for Business Central. But first, you need to prepare some additional settings for your project.

Code Analysis Tools

In the AL language, there are provided four code analysis sets. The code analyzers are responsible for checking if the code that is developed is done according to Microsoft guidelines and if the provided code would not give you problems when publishing your extension in AppSource or Business Central tenant as a modification for one customer. Below you can find names and a short description of the sets:

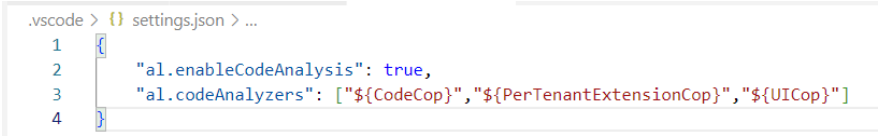
- ✓ **CodeCop** – provides the set of guidelines **(mostly as warnings)** related to code (for example if variables are used or not). It is recommended to always have this set on.
- ✓ **AppSourceCop** – provides the set of guidelines **(mostly as errors)** related to the extension if it would be published on AppSource for general download by any company that uses Business Central.
- ✓ **UICop** – provides the set of guidelines related to the user interface.
- ✓ **PerTenantExtensionCop** – provides the set of guidelines related to extensions that are developed for a particular customer.

To enable code analysis, open the **settings.json** file. You can do it either from the menu File, Preferences, and then Settings or by writing in the Command Pallet - **Open Workspace Settings (JSON)**.

To enable the code analysis, you need to add the below parameters:

- ✓ **al.enableCodeAnalysis** – set it to **true**. This setting enables code analysis functionality
- ✓ **al.codeAnalyzers** – This setting specify which analysis is needed in the project. For purpose of this workbook add: CodeCop, UICop, PerTenantExtensionCop

Your file should look at the end similar to below.



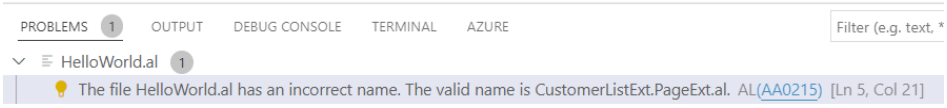
HINTS

Instead of manually typing the names of the settings and code analyzers use Ctrl+Space to get the available options.

*Action **Open Workspace Settings (JSON**) will open settings enabled only in the current project. If you want to create global settings choose action **Open Settings (JSON)**.*

Read more about code analyzers here: https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-using-code-analysis-tool

When you will save the file with settings you would be able to see that there is already one problem (warning) – the name of the file **HelloWorld.al** is incorrect.



File Naming Convention

In the AL language, one of the good practices is to create one file per object. In theory, there is a possibility that in one file is more than one object however, it is not common practice.

The file name can be created as:

<Object Name Excluding Prefix>.<Object Type>.al

In the below table, you can find the name of the most common object types.

Type of the object	Object Type in File Name
Table	Table
Table Extension	TableExt
Page	Page
Page Extension	PageExt
Codeunit	Codeunit
Report	Report
Report Extension	ReportExt
Enum	Enum
Enum Extension	EnumExt
Permission Set	PermissionSet

HINTS

There are more object types used in Business Central however you do not need to know all at this point. You can find how another object should be named here: https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/compliance/apptest-bestpracticesforalcode

AFFIXES – AVOID CONFLICTS BETWEEN EXTENSIONS

When creating new objects for your solution, each object extension needs to have its own prefix or suffix. It is mandatory to avoid conflicts between many applications in the same database. For example, if your company would create an object, for example, table "Bonus Header" and another company (or Microsoft) would create a table with the same name, then either you will not be able to install your extension or the customer will have issues when deploying other extensions.

The same situation can happen when the solution extends standard objects such as pages or tables. For each field, control, or action you need to add the prefix or suffix.

The general rule is that an affix should contain at least three characters. It is registered by the company in Microsoft.

HINTS

If you use a prefix do not add it to the file name (only keep it in the object name). It would be easier to find and navigate thru the objects in your solution.

*The affixes are mandatory for apps that are published on App Source. To enable checking the affixes create manually a new file in your project - **AppSourceCop.json**.*

Even if affixes are not mandatory for per tenant extensions it is highly recommended to use the affixes for such projects as well.

Even if affixes are not mandatory for per tenant extensions it is highly recommended to use the affixes for such projects as well

*If the company does not have any affix send an email to **d365val@microsoft.com**. It takes around one or two business days to register the affix.*

PROJECT FOLDER STRUCTURE

There are no strict rules on how to structure your solution. However, it is good to put the objects in folders to easily navigate between them. Common practice is that the top level contains two folders that you need to create manually:

- ✓ **src** or **source** – contains all files related to the functionality.
- ✓ **res** or **resources** – contains the logo of the extension.

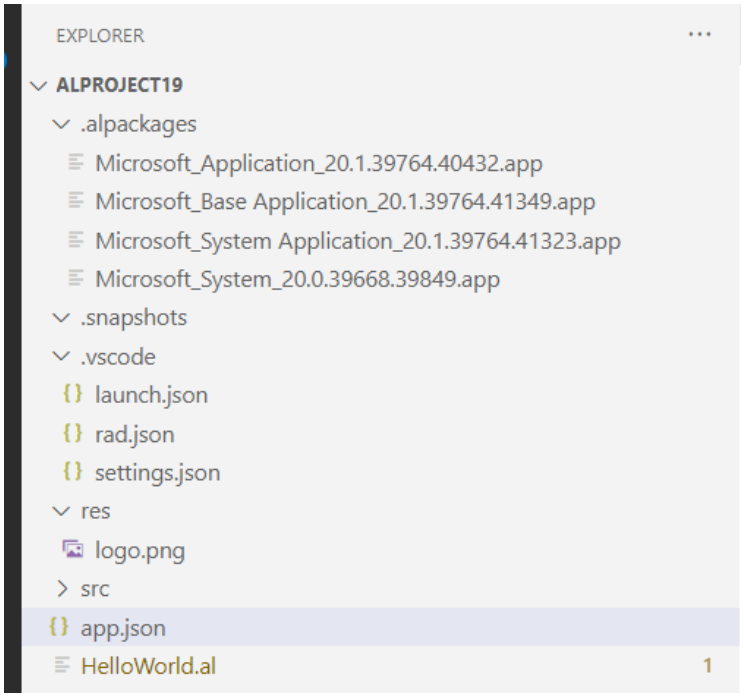
In folder **src,** you would be able to find all files created in your extension. Many companies create subfolders and group objects. They can be grouped by object type or by functionality.

TASK: UPDATE APP.JSON FILE

Now when you know which application you will create you can update the app.json file.

1. Open the **app.json** file and update properties such as **name**, **description** and **brief,** and **URL**
2. Create two folders on top-level **src** and **res**
3. Find your company logo (or any other file in jpg or png format) and copy it to the **res** folder
4. Update **app.json** file with logo path
5. Publish your application and go to Extension Management to see the results

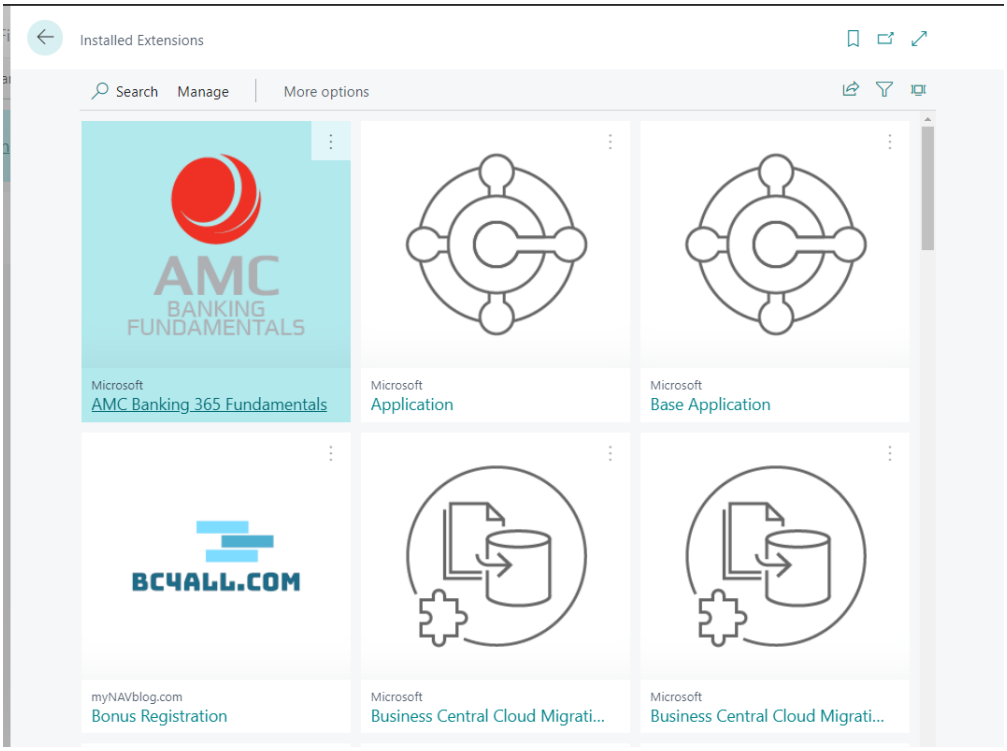
At this moment your structure of the project should look like the below.



Your **app.json** file should contain information as presented on the screen below.

```
{
  "id": "4d35d9fd-22c9-452b-89c5-cf74cd47edda",
  "name": "Bonus Registration",
  "publisher": "myNAVblog.com",
  "version": "1.0.0.0",
  "brief": "Extension allows to register bonuses for customers.",
  "description": "Extension allows to register bonuses for customers and generate reports for them.",
  "privacyStatement": "",
  "EULA": "",
  "help": "",
  "url": "https://www.myNAVblog.com",
  "logo": "res/logo.png",
  "dependencies": [],
  "screenshots": [],
  "platform": "1.0.0.0",
  "application": "20.0.0.0",
  "idRanges": [
    {
      "from": 50100,
      "to": 50149
    }
  ],
  "resourceExposurePolicy": {
    "allowDebugging": true,
```

After publishing the extension on the Extension Management page, you should see your extension with the proper name, publisher, and logo.



CHAPTER SUMMARY

- ✓ In this chapter, you found out what is your task. It will be explained in detail in the next chapters.
- ✓ You know now how to use the code analyzers.
- ✓ You know what is prefix/suffix and when you need to use it. Also, you know how to register the prefix for your extension.
- ✓ You get familiar with how you should create new file names in the project.
- ✓ You updated the app.json file with the correct information

CHAPTER 4

TABLES, ENUMS, PAGES, AND PERMISSION SETS

OBJECTIVES

In this chapter, you will develop base tables and pages for the Bonus Registration extension. The objectives are:

- ✓ Understand what object range is
- ✓ Get familiar with objects type table and page
- ✓ Know the basic type of fields
- ✓ Develop the Bonus Card
- ✓ Create Permission Set for the objects

OBJECT RANGE

Most objects in the AL language need to have a unique number per type. In other words, in the database, there cannot be two of the same objects such as a table, page, or for example codeunit with the same number. However, the table and page can have the same number.

Object range is controlled in the **app.json** file in property **idRanges**. When doing development for per tenant extension you are free to use any object range between 50000 and 99999.

When doing development for **AppSource** apps the range is assigned by Microsoft and the publisher needs to use only that specific range. It starts from 70 million.

TABLES OVERVIEW

A table is an object where you can store the data. A table in AL language contains:

- ✓ Table properties
- ✓ Set of fields
- ✓ Keys
- ✓ Global variables
- ✓ Table triggers

Each table needs to have a unique number (in the object range) and name. It means that in the database there cannot be two tables with the same name or the same number - even if the name is part of a different extension.

TABLE PROPERTIES

The table properties are added for the whole object. There are four properties that you will need to know when starting development for the AL language.

DataClassification	It is responsible for the classification of the table in terms of GDPR. It is mandatory and should be different than ToBeClassified.

Caption	Caption for the table. It should not contain the prefix or suffix.
DrillDownPageId	The name of the page will be shown when the user will use the function DrillDown on the page. For example, when clicking the calculated field.
LookupPageId	The name of the page will be shown when the user will use the function Lookup on the page. For example, when clicking more options.

TABLE TRIGGERS

The table triggers allow you to add the code directly to the table. Four table triggers tell what code is triggered when you do one of the following things.

Insert the record	When inserting the record, **OnInsert()** trigger will be executed.
Modify the record	When saving the modified record, **OnModify()** trigger will be executed.
Delete the record	When deleting the record, **OnDelete()** trigger will be executed.
Rename the record	When changing the value in any field which is in the primary key, **OnRename()** trigger will be executed.

HINTS

If you do not use any of the triggers in your table then remove it from the code so it is more readable for everyone.

TABLE FIELDS

The fields in the table can have different types. Therefore, you do not need to be worried if a field that is defined as a numeric field (integer or decimal), can store also other characters than numbers.

There is quite a long list of field types, but only a few are needed to begin the development. In the below table, you can find types that are used in most cases.

The normal class fields, like the table, have properties. For each field type you can, with the property **Editable**, decide if a field is editable or not. Two properties are mandatory - **DataClassification** and **Caption**. The caption of the field will be used as the default caption when you use that field. The **DataClassification** property is needed for GDPR and must be different than **ToBeClassified**. In the table below you will find the properties which are good to be considered when adding the field to your extension.

Code	It is an **alphanumeric** field that is automatically converted to uppercase. It is used for storing mostly unique values (very often as primary key) of the record such as customer numbers or code in a dictionary. It has got, in most cases, 20 or 10 characters.	Setting **NotBlank** property will force you to put not empty value in the field. Adding the **TableRelation** property will allow you to Lookup the values from a different table.
Text	It is a text field that is used to store such things as descriptions or addresses. In most cases, it is a length of 100 characters but can store up to 2048 chars. A common practice is to create the second field for description if 100 chars are not enough.	
Date	It stores the date. Automatically it will show the calendar to choose the date.	
Integer	It is a numeric field that stores whole numbers.	Setting **BlankZero** property will show an empty value instead of zero. Setting **MinValue** and **MaxValue** parameters will force you to use only values in the range.
Decimal	It is a numeric field that stores decimal numbers.	Setting the **DecimalPlaces** property will allow you to define the minimum and the maximum number of decimal places. You can use the same properties as the **Integer** type.

Enum	This field type shows the list of options defined in the separate object - Enum. When creating the field, you need to specify which Enum you would like to use.

HINTS

Each field that is added to the table needs to have a unique number. In your tables start the number from 1. You can see the number of the field.

Do not add affixes to the fields in newly created tables. The affix for the table as the object is enough.

Some of the fields are added automatically to the table without development. Those fields are systemId, SystemCreatedAt, SystemCreatedBy, SystemModifiedBy, SystemModifiedAt.

FIELD TRIGGERS

The field triggers allow you to add the code directly to the field. There are two triggers assigned to the field. The first one - OnLookup(), allows you to add the code when a user clicks on the page lookup function. However, the second one is more important. It triggers when the user will put value in the field. It is called OnValidate().

FLOW FIELDS – A SPECIAL CLASS OF FIELDS

The **Flow Fields** are a special class of fields. You can set them with the property **FieldClass**. For those fields, you cannot set **DataClassification**.

It allows to do easy mathematic operations, such **as count, sum, or get maximal** or **minimal value**. It also allows you to **show value from a different table** (lookup) or **check if the record exists**. To tell what the Flow Field should show, you can use the property **CalcFormula**.

For the Flow Fields, you should always set the editable property to false.

Table Keys

In the table, you need to define at least one key which will be the Primary Key. It means that it will not be allowed to insert the same value twice in the field. You can, in an easy way, get the record in code by using the method **Get()** and specifying the value of the primary key.

Some of the tables can have more than one field in the Primary Key. It means that there cannot be two records with exactly the same value in all the fields included in the key.

Hints

*A table can have more than one key which allows for increased performance. By default, the Primary Key has the name **PK**.*

*Many tables (such as Customer, Vendor, or Item) have a No. field as the primary key. This field can have alphanumeric values since is a type of **Code[20]**. To get automatically the next available value Number Series are used.*

*Some of the tables (for example Log tables) have the Primary Key set as Entry No. which is field type **Integer**. To automatically get the next available number, the field can have special property added – **AutoIncrement.***

Task: Update App.json file

To create a new object, you need to set the range which will the object use.

1. Open the **app.json** file and update property **idRanges**. Set **from** object range 65400 and **to** 65600.

Your **app.json** file should contain information as presented on the screen below.

```
16  "application": "20.0.0.0",
17  "idRanges": [
18    {
19      "from": 65400,
20      "to": 65600
21    }
22  ],
23  "resourceExposurePolicy": {
```

TASK: CREATE APPSOURCECOP.JSON

AppSourceCop.json file control which affixes are used in the project. To avoid warnings related to the name of the objects that will be created do below:

1. Create in top folder file **AppSourceCop.json** and add to it parameter **mandatoryAffixes**.
2. Use "**MNB**" as a mandatory affix.
3. Open Command Pallet (F1) and run function **Developer: Reload Window**.

Your **AppSourceCop.json** file should contain information as presented on the screen below.

TASK: CREATE A BONUS HEADER TABLE

Your Bonus Card will be created from two tables. The first one will be a header, where you will store the information about the customer, status, and starting and ending date. In the second table, you will store information about the granted bonus.

1. Create a new file **BonusHeader.Table.al** in folder **src**. You may also create a subfolder **Bonus**.
2. Use a snippet **ttable** (double t in the beginning) to create a new table. Add table number and table name "**MNB Bonus Header**".

HINTS

Snippets allow you to create faster the code. They started with the letter "t" for example to create a new table start writing ttable and then the whole structure of the object would be created automatically. Using the Tab key, you can switch to the next place where you need to change the values.

To get automatically the number of the object (next available) click Ctrl+Space. It will show you the next number of the object.

When adding a caption to the Bonus Header do not add the word Header.

3. Create a new field **"No."** - use type Code[20]. Remember about properties. This field will be the primary key.

   Create a new field **"Customer No."** - use type Code[20]. Remember about properties and add **TableRelation** to table "Customer"

   ### HINTS

   When you specify table relation you do not need to put the field to which you create a relation if this is a Primary Key and this key contains only one value. Otherwise, you need not only specify the table name but also the field. In the Customer table, there is only one field in a primary key so you do not need to add the No. field in the relation.

4. Create a new field **"Starting Date"** - use the type Date
5. Create a new field **"Ending Date"** - use the type Date

HINTS

If you set properly code analyzers you may see an error that the Table is missing a matching permission set.

This will be solved in the next pages.

SOLUTION

```
table 65400 "MNB Bonus Header"
{
    Caption = 'Bonus';
    DataClassification = CustomerContent;

    fields
    {
        field(1; "No."; Code[20])
        {
            DataClassification = CustomerContent;
            Caption = 'No.';
        }
        field(2; "Customer No."; Code[20])
        {
            DataClassification = CustomerContent;
```

```
            Caption = 'Customer No.';
            TableRelation = Customer;
        }
        field(3; "Starting Date"; Date)
        {
            DataClassification = CustomerContent;
            Caption = 'Starting Date';
        }
        field(4; "Ending Date"; Date)
        {
            DataClassification = CustomerContent;
            Caption = 'Ending Date';
        }
    }
    keys
    {
        key(PK; "No.")
        {
            Clustered = true;
        }
    }

}
```

Permission Sets

Permission Set is a special object type that defines which permissions are included in it. It is possible to add specific objects to it (such as tables, codeunits, pages, etc.) or other permission sets.

Parameter **Assignable** allows specifying if the Permission set can be assigned directly to the users.

When adding the permission to the object it is possible to define if permission allows executing it (X), or in the case of table data if the user would have permission to read (R), modify (M), insert (I), delete (D).

Hints

Not all objects require assignment in permission sets. One of such objects is Enums which you will create in the next steps.

*There is a difference between adding permission to a table as an object (then it is possible to assign only Execution permission) or to **table data** (then it is possible to add Read, Insert, Modify and Delete permissions.*

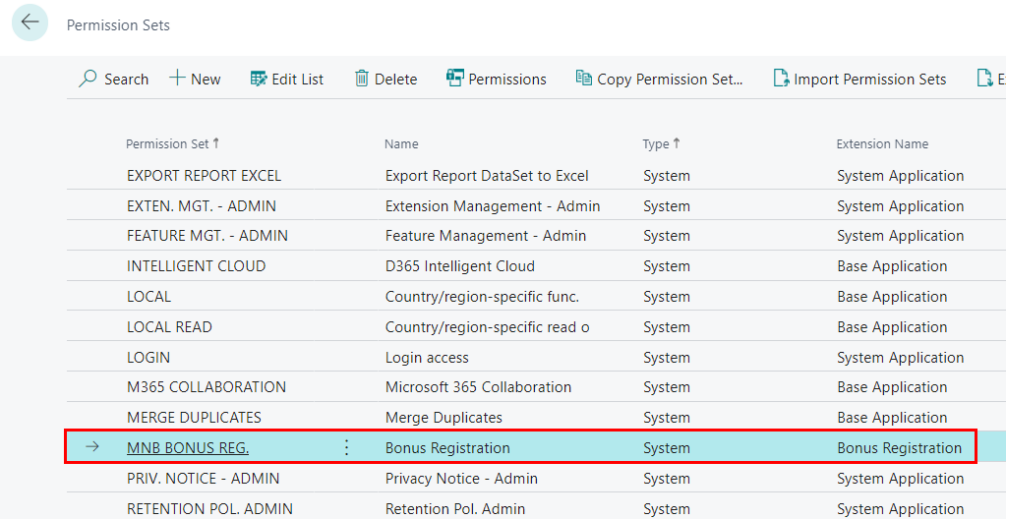
TASK: PERMISSION SET

To remove the error related to the missing permission for the Bonus Header table add a new permission set.

1. Create a new subfolder in the **src** folder and name it: **Permissions**.
2. Create a new file **BonusReg.PermissionSet.al** in the Permissions folder.
3. Use a snippet **tpermissionset** to create a new permission set. Add number and name "**MNB Bonus Reg.**".
4. Set **Assignable** property to true and add **Caption** property **Bonus Registration**
5. Add Read, Insert, Modify and Delete permission to table data **"MNB Bonus Header"**
6. Publish the extension and check if the permission set exists

SOLUTION

```
permissionset 65400 "MNB Bonus Reg."
{
    Caption = 'Bonus Registration';
    Assignable = true;
    Permissions =
        tabledata "MNB Bonus Header" = RMID;
}
```



ENUMS

Enum is the object type that allows specifying the static (not defined by the user) options. It is possible to allow to extend the enum. For that property, **Extensible** should be set to true.

Each value of the enum has a number and name. Additionally, the values should have **Caption** property.

HINTS

Values in enums start from 0.

If you want to show an empty option add to the enum value with the name None and with Caption = ' ' (space between the single quote).

Remember to ad affix to the enum object. The values do not need the affix.

TASK: BONUS HEADER STATUS ENUM

To track the status of the Bonus the new enum is needed with values **Open** and **Released**.

1. Create file **BonusHeaderStatus.Enum.al** in the same folder as Bonus Header Table.
2. Use a snippet **tenum** to create a new enum. Add number and name "**MNB Bonus Header Status**".
3. Set property **Extensible** to false.

4. Add two values to the enum – **Open**, **Released**.

Solution

```
enum 65400 "MNB Bonus Header Status"
{
    Extensible = false;
    value(0; Open)
    {
        Caption = 'Open';
    }
    value(1; Released)
    {
        Caption = 'Released';
    }
}
```

Task: Add the Status field to the Bonus Header

The new field with the status needs to be added to the **Bonus Header** table.

1. In the **Bonus Header** table add new field **Status**. Set the type to Enum and specify the enum that was created in the previous task.
2. Remember about mandatory properties such as **DataClassification** and **Caption**

Solution

```
field(5; Status; Enum "MNB Bonus Header Status")
        {
            DataClassification = CustomerContent;
            Caption = 'Status';
        }
```

TASK: CREATE A BONUS LINE TABLE

The Bonus Line table will store information about granted bonus percent. It will relate to the header by using **Document No.** field. Users will be able to put bonuses either for all items or for one specified item. If a user would choose an item, it will be possible to choose a number from the list.

This table will have multiple fields in the primary key.

1. Create new file **BonusLineType.Enum.al** in the same folder as Bonus Header table and create new enum "MNB Bonus Line Type"
2. Create value 0 with the Caption **"All Items"**
3. Create value 1 with the Caption **"Item"**

SOLUTION

```
enum 65401 "MNB Bonus Line Type"
{
    Extensible = false;
    value(0; "All Items")
    {
        Caption = 'All Items';
    }
    value(1; "Item")
    {
        Caption = 'Item';
    }
}
```

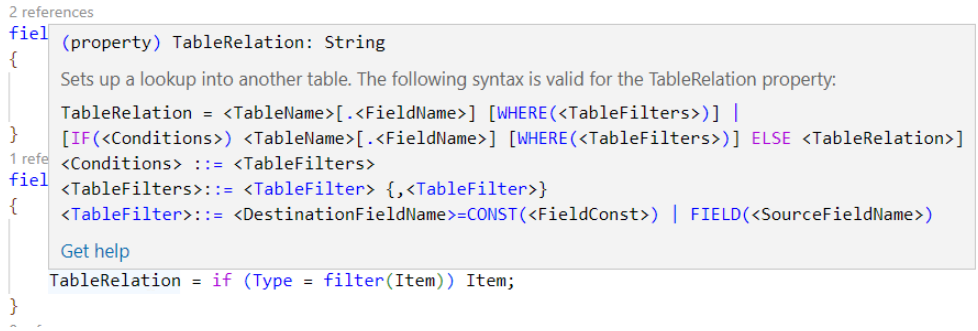
4. Create a new file **BonusLine.Table.al** in the same folder as the Bonus Header table
5. Use a snippet **ttable** to create a new table. Add table number and table name **"MNB Bonus Line"**
6. Create a new field **"Document No."** - use type **Code[20]**. Remember about properties and add **TableRelation** to **"MNB Bonus Header"**

7. Create a new field **"Type"** - use type Enum. Remember about properties and add Enum **"MMNB Bonus Line Type"**
8. Create a new field **"Item No."** - use type **Code[20]** and add **TableRelation** to **"Item"** but only if Type is Item

HINTS

It is possible to add conditions in table relation. For example, if one of the fields (for example Partner Type) has the value Vendor then the Table Relation points to the Vendor table and if the value is Customer it points to the Customer table.

To get more information on how to do it hover on property TableRelation and you will see the syntax.



9. Create a new field **"Bonus Perc."** - use type Integer. Allow only values from 0 to 100
10. Add fields **"Document No.", "Type",** and **"Item No."** to the primary key. This table will have the primary key that contains 3 fields
11. Add the table to the already existing permission set

SOLUTION

```
table 65401 "MNB Bonus Line"
{
    DataClassification = CustomerContent;
    Caption = 'Bonus Line';

    fields
    {
```

```
        field(1; "Document No."; Code[20])
        {
            DataClassification = CustomerContent;
            Caption = 'Document No.';
            TableRelation = "MNB Bonus Header";
        }
        field(2; Type; Enum "MNB Bonus Line Type")
        {
            DataClassification = CustomerContent;
            Caption = 'Type';
        }
        field(3; "Item No."; Code[20])
        {
            DataClassification = CustomerContent;
            Caption = 'Item No.';
            TableRelation = if (Type = filter(Item)) Item;
        }
        field(4; "Bonus Perc."; Integer)
        {
            DataClassification = CustomerContent;
            Caption = 'Bonus Perc.';
            MinValue = 0;
            MaxValue = 100;
        }
    }

    keys
    {
        key(PK; "Document No.", Type, "Item No.")
        {
            Clustered = true;
        }
    }
}
```

SOLUTION

```
permissionset 65400 "MNB Bonus Reg."
{
    Caption = 'Bonus Registration';
    Assignable = true;
    Permissions =
        tabledata "MNB Bonus Header" = RMID,
```

```
        tabledata "MNB Bonus Line" = RMID;
}
```

PAGES OVERVIEW

On pages, users interact with data. They can insert, modify or delete data. Additionally, they can run custom actions.

A page in the AL language contains:

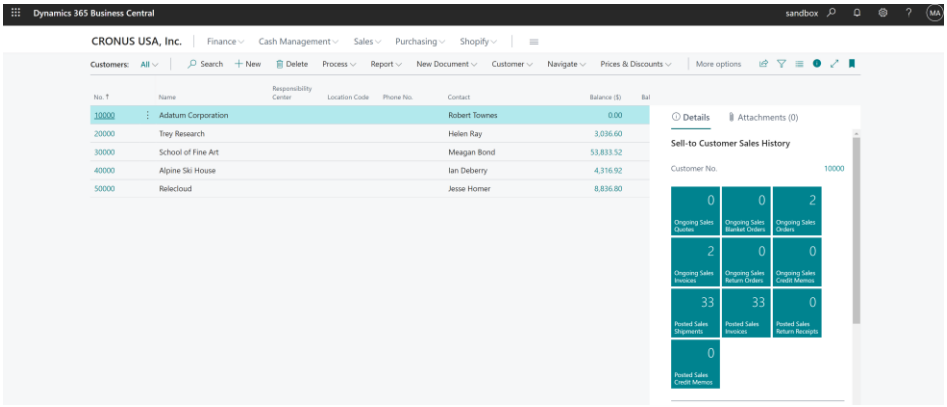
- ✓ Page properties
- ✓ Set of controls
- ✓ Set of actions
- ✓ Global variables
- ✓ Page triggers

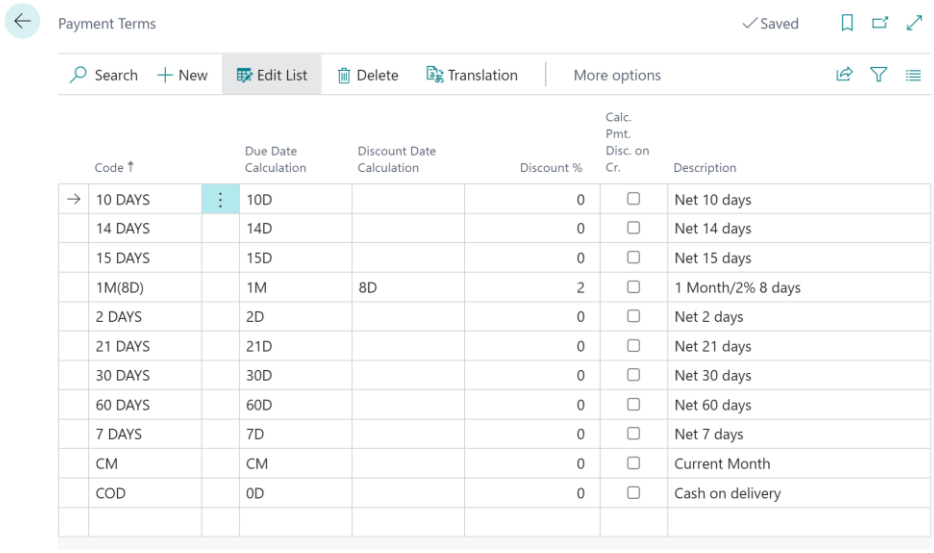
In Business Central there are different types of pages. The most important are explained below.

LIST PAGE

The list page is used when more than one record needs to be shown on the page. This kind of page is, in most cases, visible from the menu or Tell Me functionality. For data such as customers, vendors, or purchase orders (so data from master tables and transactional data) lists are not editable. The lists which present dictionaries, for example, payment terms or user setup, are editable.

Examples of pages with the type List you can find below.





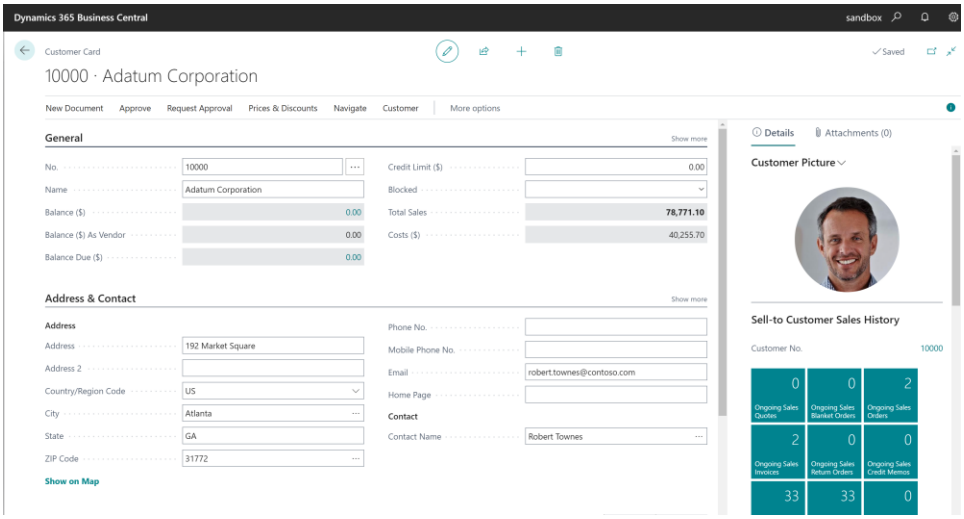
Code	Due Date Calculation	Discount Date Calculation	Discount %	Calc. Pmt. Disc. on Cr.	Description
10 DAYS	10D		0	☐	Net 10 days
14 DAYS	14D		0	☐	Net 14 days
15 DAYS	15D		0	☐	Net 15 days
1M(8D)	1M	8D	2	☐	1 Month/2% 8 days
2 DAYS	2D		0	☐	Net 2 days
21 DAYS	21D		0	☐	Net 21 days
30 DAYS	30D		0	☐	Net 30 days
60 DAYS	60D		0	☐	Net 60 days
7 DAYS	7D		0	☐	Net 7 days
CM	CM		0	☐	Current Month
COD	0D		0	☐	Cash on delivery

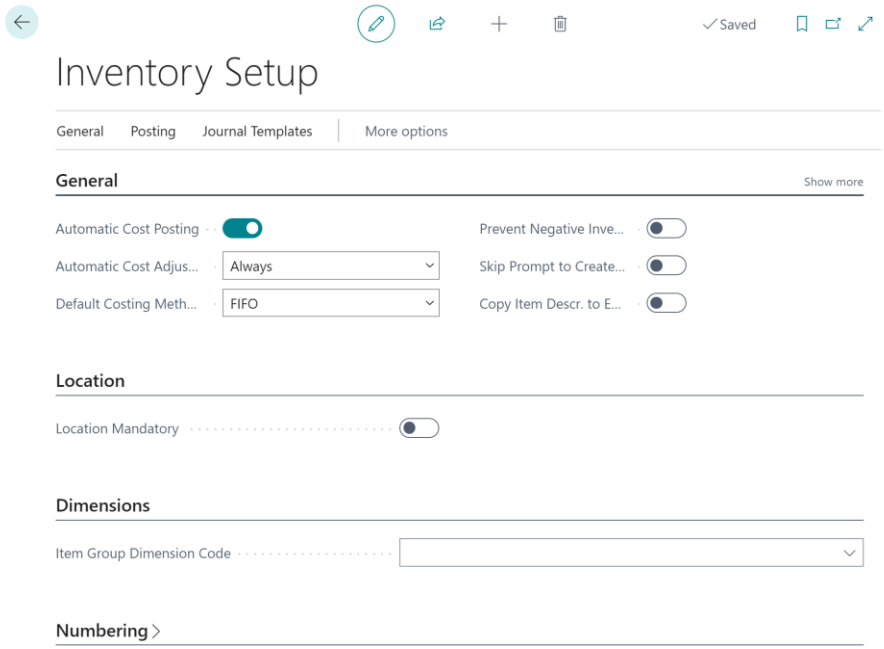
CARD PAGE

The card page is used when only one record needs to be shown on the page. This type is used, in most cases, for the master data. For example, a customer, a vendor, or an item. Such pages are, in general, editable but cannot be accessed from the menu. Only from the list page for master data.

The setup pages, such as Inventory Setup are also the card pages. Those can be accessed from the menu or Tell me functionality, and are editable but a user cannot delete or insert a record directly.

Examples of the pages with the type Card you can find below.



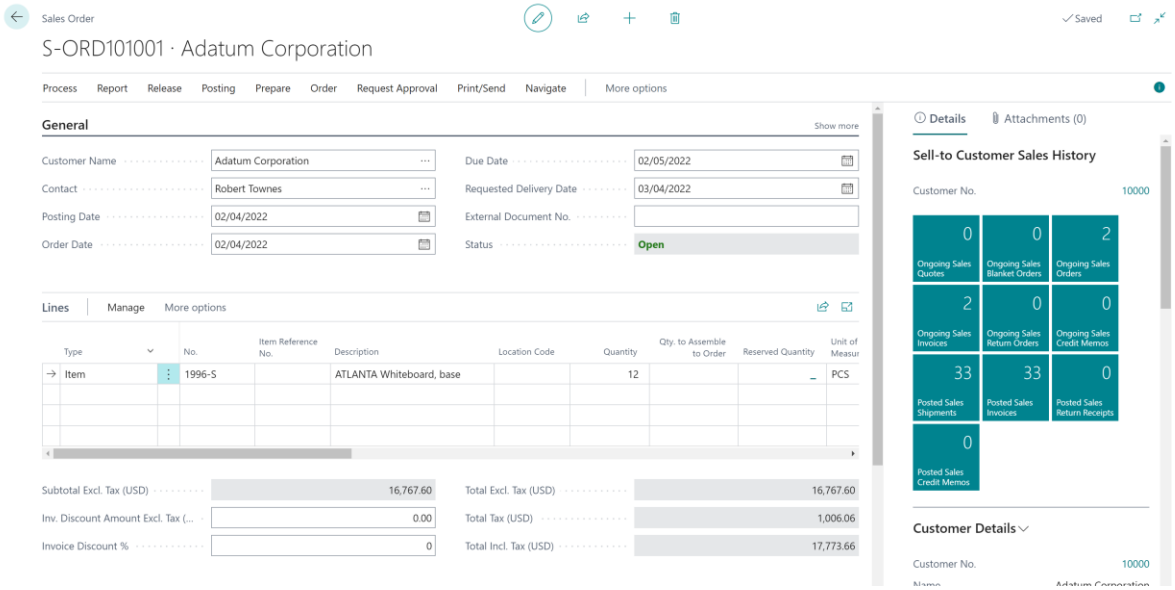


DOCUMENT PAGE

The document page is used when there should be a header and lines presented. This type is used, in most cases, for the documents such as orders, invoices, and posted invoices.

In practice, such a page is created using two separate pages. The first one is with the type Document where data from the header table is presented. The second one presents lines and is created with the type **ListPart**. Then such a page is added to the Document page as a part.

An example of a document page with the ListPart page you can find below.



PAGE PROPERTIES

Each page has its own properties. There are common properties that can be used on all types of pages. In the below table, you will find the most useful properties. Some of the properties are only for the specific type of page. If so then it is stated in the table.

PageType	This property defines the type of page. The most common types you can find above.
Caption	Caption for the page. It should not contain the prefix or suffix.
SourceTable	The property describes from which table data is presented on the page. It can be only one table set per page.
UsageCategory	If the page should be visible from **Tell Me** functionality, this property is mandatory. Additionally, you will need to fill the **ApplicationArea** property if you want that page will be seen in the Tell Me. The **ApplicationArea** property describes in which areas of the system the page is visible.
Editable	With this property, you can tell if the page is editable or not. By default, it is editable.
DeleteAllowed, InsertAllowed, ModifyAllowed	The purpose of those three properties should be rather clear. They tell if the page should allow delete, insert or

	modify records. By default, those properties are set to true.
CardPageId	This property is only for pages with a type list. You can decide which page will be open as a card page for the list.

PAGE TRIGGERS

Like the table, there are a few triggers on the page. Not all are used so often. That is why, in this workbook, only the most important will be described. In triggers, you can write a code that will be run when triggered.

Open the page	When opening the page, **OnOpenPage()** trigger will be executed.
Record is retrieved from the database	When putting the focus on the record, **OnAfterGetCurrRecord()** trigger will be executed.

HINTS

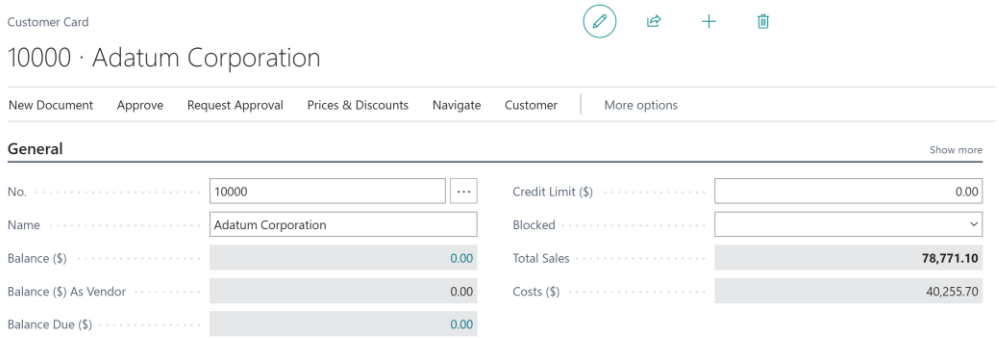
It is not common to put code in triggers related to inserting, modifying, or deleting the data on the page.

Rather the code is put on the table directly.

PAGE CONTROLS

On each page in the layout section, you will find the controls. Controls are assigned to one of two areas - **Content** or **FactBox**.

In the Content area, you will find the fields which should be shown on the page. Those fields are grouped in **FastTabs** (in the case of card and document pages) or the repeater (in the case of list pages). An example can be General FastTab.



HINTS

You cannot control where exactly on the screen the control (field) will be displayed. Only in which group (Fast) and in which order

It is not often to put the captions on pages. If you put the caption on the table then it will be used on all pages.

*It is possible to add code to fields on the pages as well. There are triggers **OnValidate()** and **OnLookup()** but it is more common to put code directly on the table.*

Fields have their properties. For now, only two are very important. The first is **ApplicationArea**. It describes in which area of the system field should be shown. The second one is the **ToolTip**. It gives users basic help in the field.

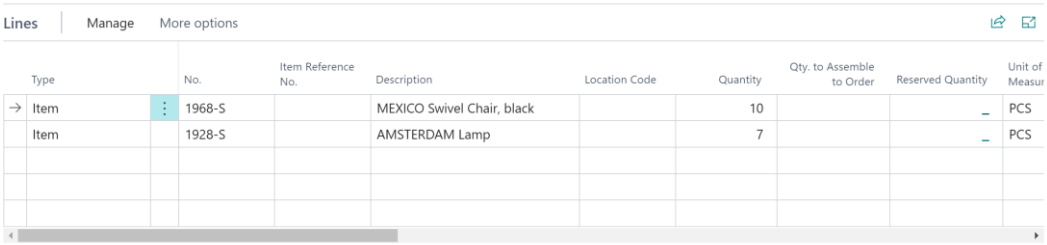
HINTS

*If you would forget to add **ApplicationArea** control will be not shown on the field. When doing development, you can set this property to All.*

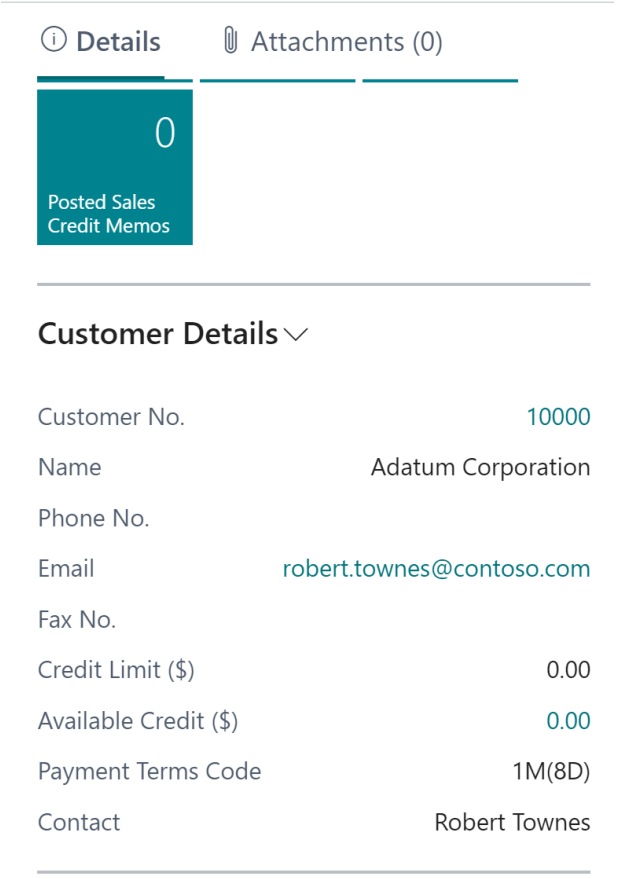
***ToolTip** is not mandatory to add on the control but it is good practice to add it. on controls (fields) **ToolTip** should start from the word Specifies…*

It is also possible to add a part control. It will allow you to embed another page on your main page. You should put the same properties as the normal field. Also, define what is the page name, that

should be added. Additionally, specify the link between the pages (for example the same document number). Examples of the parts are lines for a bonus card or a purchase order.



In the **FactBox** area, you can show parts the same way as in the Content area. FactBox is shown on the right side of the page. It is very common to use a FactBox to show more data about the record which is present in the main window. Normally the page which is present in the FactBox has a special type of page - **CardPart**. An example of the FactBox can be Customer Details on Sales Order.



HINTS

*You cannot control **FactBox** width or size and place where it is shown (it is not possible to move it to the left).*

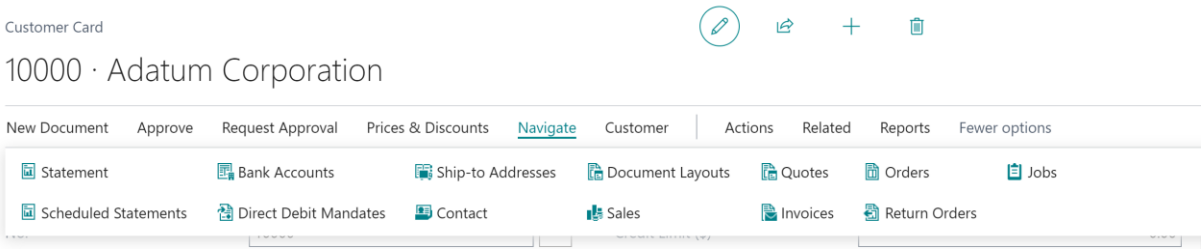
Also, you cannot control if the FactBox will be shown in Details or Attachments on the screen.

You can add many FactBoxes to the same page.

PAGE ACTIONS

On each page, you can add special actions. The actions are visible in the top part of the page. You can decide, what will happen when the user will click the action. It can be that the new page will be opened, the report will be run, or some code will be triggered.

Actions are grouped in the areas. There are four areas that you can choose: **Creation**, **Navigation**, **Processing**, and **Reporting**. Actions that are used to open related data, should be placed in the Navigation area. Those who do some action, for example posting a document, or changing status, should be placed in the Processing area. Related reports should be placed in the Reporting area.



Actions, the same as controls, need some properties to work correctly. In the table below you can find the properties needed for actions.

ApplicationArea	This property is mandatory. It says in which area of the system action will be visible.
Caption	Caption for the action. It should not contain the prefix or suffix.

Image	The property tells what the image for the action is.
Promoted	If action is set as promoted, then it is shown on the **Home** tab. If you want that action to be shown only in the Home tab, then set the property **PromotedOnly** to true. When action is promoted, then you can put it in the proper category using the property **PromotedCategory**. There are three standard categories: New, Process, and Report.
RunObject	If you want to run the object, for example, another page or report, you need to specify the object type and name. The link between your page and the object can be set in the property **RunPageLink**. In case you use the **RunObject** property do not write code in trigger **onAction()**.
ToolTip	ToolTip for the action is mandatory if your extension would be published on AppSource. It gives the user quick help with what the action does.

Actions, has also a trigger – **OnAction()**. To be able to see the action on the page it must contain either the **RunObject** property or code in the **OnAction** trigger.

HINTS

It is good practice to assign an image to action. You can see how the image looks directly in the Visual Studio Code.

TASK: UPDATE APP.JSON

Before creating pages you need to update the **app.json** file and add one parameter that will help create pages – **NoImplicitWith**.

1. Open the **app.json** file add new property **features** to it add **NoImplicitWith**

Your **app.json** file should contain information as presented on the screen below.

```
23    "resourceExposurePolicy": {
24      "allowDebugging": true,
25      "allowDownloadingSource": false,
26      "includeSourceInSymbolFile": false
27    },
28    "features": ["NoImplicitWith"],
29    "runtime": "9.0"
30  }
```

HINTS

This property is related to legacy code therefore With statement will not be described in this workbook. To avoid using this statement in the code parameter in app.json needs to be enabled.

TASK: CREATE BONUS LIST PAGE

The **Bonus List** page will present all available, at this moment, fields from the **Bonus Header** table. Also, you will add new action which will open the **Customer Card** page for the customer specified in the bonus.

1. Create a new file **BonusList.Page.al** and create a new page "**MNB Bonus List**" using snippet **tpage** and choosing list page

   ### HINTS

   A few snippets are creating a page remember to choose the proper type to have less work.

2. Add page properties for **Caption** (choose Bonuses), **UsageCategory** – (choose List) and **ApplicationArea** choose (All)
3. Make sure that the page is not editable, and the source table is "**MNB Bonus Header**"
4. Add all fields to the page from the table. Remember to add **ToolTip** and **ApplicationArea** for all fields
5. Add new action "**Customer Card**" in the **Navigation** area. Action should open the customer card for the customer specified in the "Customer No." field. To create an action, you can use a snippet taction. Remember about the **Image**

HINTS

The action also needs to have some unique name. Common practice is to name it similar to caption but without the space.

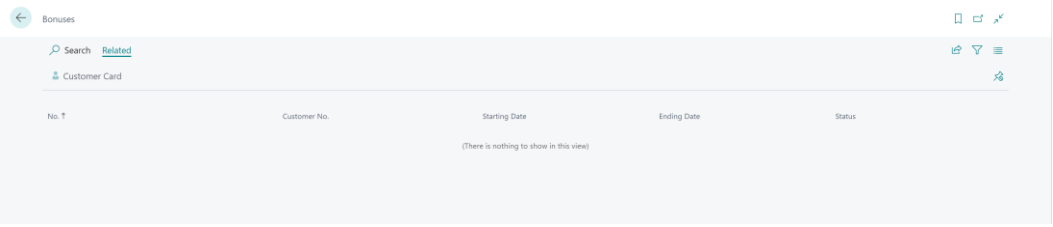
SOLUTION

```
page 65400 "MNB Bonus List"
{
    PageType = List;
    Caption = 'Bonuses';
    ApplicationArea = All;
    UsageCategory = Lists;
    SourceTable = "MNB Bonus Header";
    Editable = false;

    layout
    {
        area(Content)
        {
            repeater(Control1)
            {
                field("No."; Rec."No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies the bonus number.';
                }
                field("Customer No."; Rec."Customer No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies the customer number.';
                }
                field("Starting Date"; Rec."Starting Date")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies the starting date.';
                }
                field("Ending Date"; Rec."Ending Date")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies the ending date.';
                }
                field(Status; Rec.Status)
                {
                    ApplicationArea = All;
```

```
                    ToolTip = 'Specifies the bonus status.';
                }
            }
        }
    }
    actions
    {
        area(Navigation)
        {
            action(CustomerCard)
            {
                ApplicationArea = All;
                Caption = 'Customer Card';
                ToolTip = 'Open customer card for the bonus.';
                Image = Customer;
                RunObject = page "Customer Card";
                RunPageLink = "No." = field("Customer No.");
            }
        }
    }
}
```

Hints

You can publish your extension to see the changes.



Task: Add to table DrillDown and lookUp Page Id

To connect the list page with a table when even someone will click the calculated field based on the **Bonus Header** table, it is needed to add **DrillDownPageId** and **LookUpPageId** properties to the table.

1. Open the file **BonusHeader.Table.al** and assign **"MNB Bonus List"** to the **DrillDownPageId** and **LookUpPageId** properties

Solution

```
table 65400 "MNB Bonus Header"
{
    Caption = 'Bonus';
    DataClassification = CustomerContent;
    DrillDownPageId = "MNB Bonus List";
    LookupPageId = "MNB Bonus List";
```

Task: Create Bonus Card Page

The **Bonus Card** page will present all available, at this moment, fields from the **Bonus Header** table. Also, you will add new action which will open the **Customer Card** page for the customer specified in the bonus.

Page, which you will create, will be a type card and will be editable.

1. Create a new file **BonusCard.Page.al** and create a new page **"MNB Bonus Card"** using snippet **tpage.** Choose a card page template. Make sure to change the type to **Document**.
2. Add page property for **Caption** and set **UsageCategory** to **None**
3. Make sure that the page source table is **"MNB Bonus Header"**
4. Create a group **General** in the Content area. Add all fields to it from the table. Remember to add **ToolTip** and **ApplicationArea** for all fields
5. Copy the action **"Customer Card"** from **"MNB Bonus List"** and add it to the **"MNB Bonus Card"**

Solution

```
page 65401 "MNB Bonus Card"
{
    PageType = Document;
```

```
    SourceTable = "MNB Bonus Header";
    Caption = 'Bonus Card';
    UsageCategory = None;


    layout
    {
        area(Content)
        {
            group(General)
            {
                Caption = 'General';
                field("No."; Rec."No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies bonus number.';
                }
                field("Customer No."; Rec."Customer No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies bonus customer number.';
                }
                field("Starting Date"; Rec."Starting Date")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies bonus starting date.';
                }
                field("Ending Date"; Rec."Ending Date")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies bonus ending date.';
                }
                field(Status; Rec.Status)
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies bonus status.';
                }
            }
        }
    }
    actions
    {
        area(Navigation)
        {
            action(CustomerCard)
            {
                ApplicationArea = All;
                Caption = 'Customer Card';
```

```
                ToolTip = 'Open customer card for the bonus.';
                Image = Customer;
                RunObject = page "Customer Card";
                RunPageLink = "No." = field("Customer No.");
            }
        }
    }
}
```

Task: Connect Bonus List page with Card page

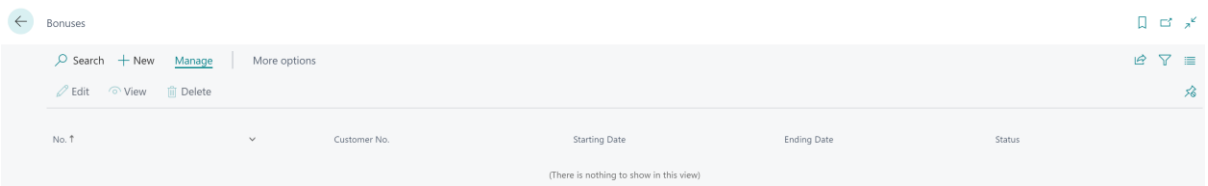
To connect the Bonus List and Bonus Card you need to specify the CardPageId property on the List.

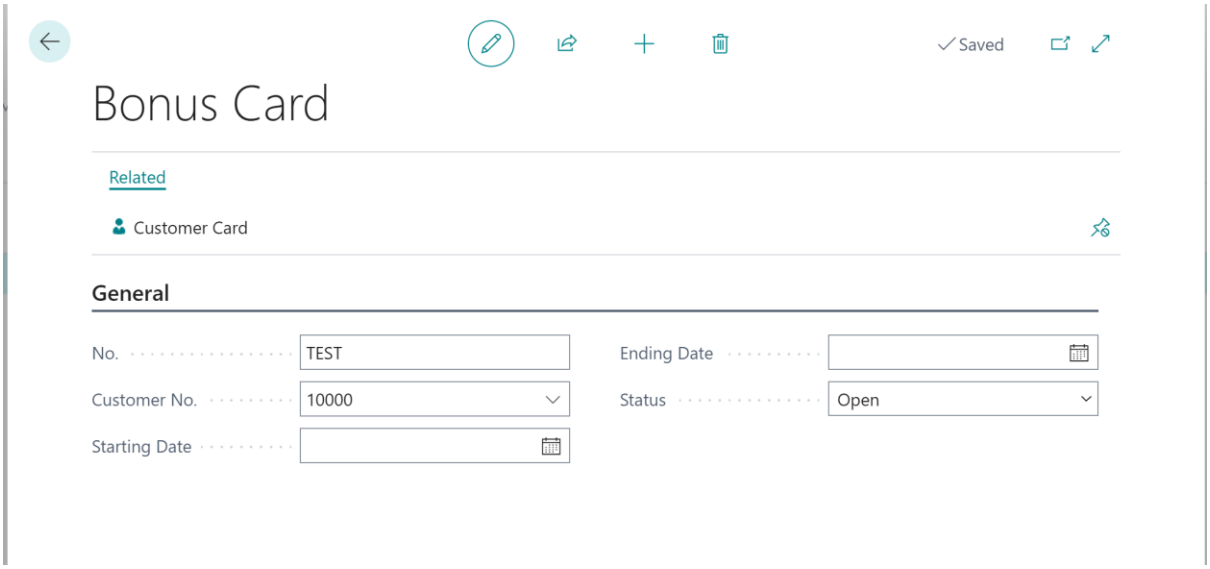
Page, which you will create, will be a type card and will be editable.

1. Go to the file **BonusList.Page.al** and add the property **CardPageId**. Choose **"MNB Bonus Card"** as the card page
2. Publish the extension and add a new bonus. Check if actions New, Edit, and Delete are working properly

Solution

```
page 65400 "MNB Bonus List"
{
    PageType = List;
    Caption = 'Bonuses';
    ApplicationArea = All;
    UsageCategory = Lists;
    SourceTable = "MNB Bonus Header";
    Editable = false;
    CardPageId = "MNB Bonus Card";
```





Hints

At this moment the number field needs to be populated manually. You can add any value in this field

Task: Create Bonus Subform Page

The **Bonus Card** page, at this point, has only a header. You need also to add the lines to define the bonus rules. For that new page with the type **ListPart** is needed. Later you will need to add the page, as a part, to the **"MNB Bonus Card"** page.

1. Create a new **file BonusSubform.Page.al** and create a new page **"MNB Bonus Subform"** using snippet **tpage** and choose a list page template. Make sure to change the type to **ListPart**
2. Add page property for **Caption** - Lines
3. Make sure that the page source table is **"MNB Bonus Line"**
4. Add fields "**Type", "Item No."** and **"Bonus Perc."**. Remember to add the **ApplicationArea** and **ToolTip** properties

Solution

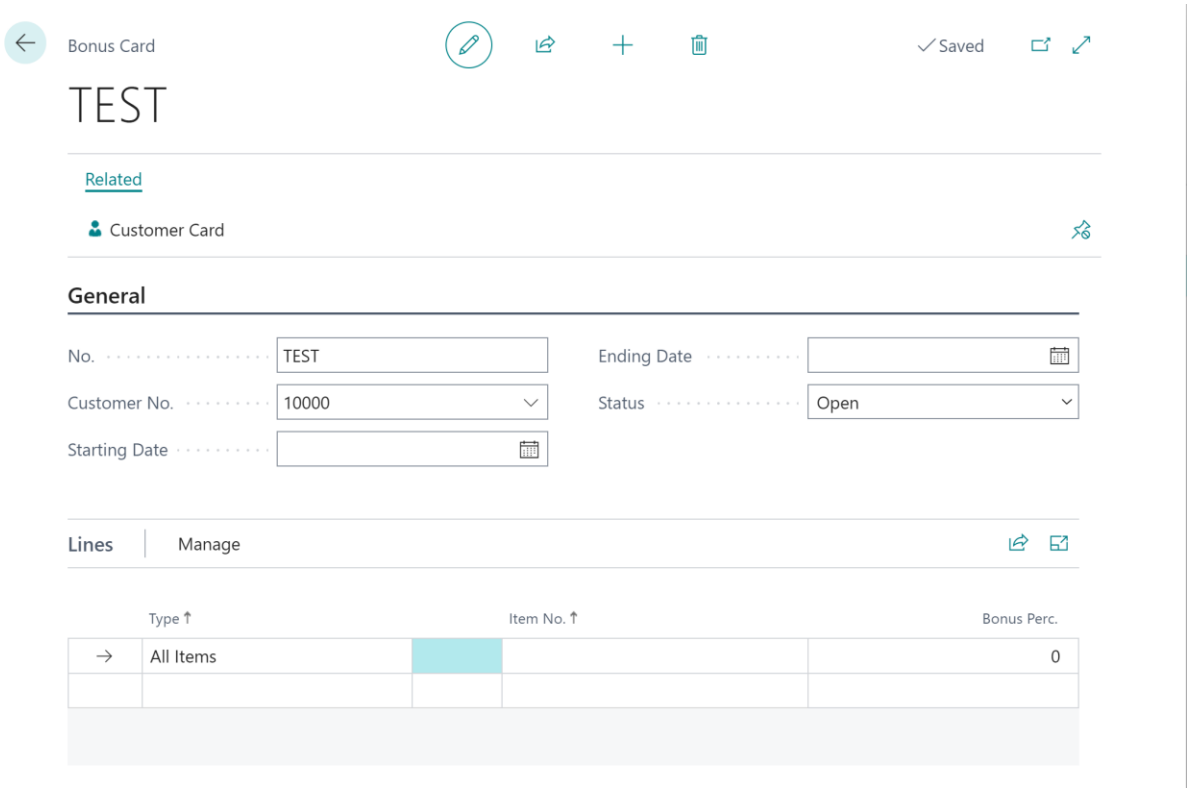
```
page 65402 "MNB Bonus Subform"
{
    PageType = ListPart;
```

```
    SourceTable = "MNB Bonus Line";
    Caption = 'Lines';

    layout
    {
        area(Content)
        {
            repeater(Lines)
            {
                field(Type; Rec.Type)
                {
                    ApplicationArea = All;
                    Tooltip = 'Specifies the type of the bonus
assigned.';
                }
                field("Item No."; Rec."Item No.")
                {
                    ApplicationArea = All;
                    Tooltip = 'Specifies item number for which bonus is
assigned.';
                }

                field("Bonus Perc."; Rec."Bonus Perc.")
                {
                    ApplicationArea = All;
                    Tooltip = 'Specifies bonus percent.';
                }
            }
        }
    }
}
```

5. Open the file **BonusCard.Page.al** and in the Content area add a new part with the name **Lines**
6. Set that the part shows page **"MNB Bouns Subform"** and add a link between the header and lines that **"No."** in the header is the same as **"Document No."** in the lines
7. Publish your code to see how the Bonus Card looks now

SOLUTION

```
        part(Lines; "MNB Bonus Subform")
        {
            ApplicationArea = All;
            Caption = 'Lines';
            SubPageLink = "Document No." = field("No.");
        }
```



CHAPTER SUMMARY

- ✓ In this chapter, you found what are tables and pages
- ✓ You developed the first tables – Bonus Header and Lines, Enum, and Permission Set.
- ✓ You also developed pages to show the Bonuses on the list and also on the card

CHAPTER 5

TABLE AND PAGE EXTENSIONS

OBJECTIVES

In this chapter, you will get information on how to add new fields to the standard tables. Also, how to show new fields on standard pages. The objectives are:

- ✓ Get familiar with objects type table extension and page extension
- ✓ Develop the table extension for the Customer table
- ✓ Develop the new action for the Customer Card

STANDARD OBJECTS

Business Central contains standard objects such as tables, pages, and more. It is not allowed to modify them directly (for example by opening them and changing the captions, adding fields, etc.). However, most tables and pages can be modified using table or page extensions.

HINTS

*To not allow other extension publishers to extend objects it is needed to add **Extensible** property to it. By default, it is possible to extend the object.*

TABLE EXTENSION OVERVIEW

A table extension is an object which allows you to modify the standard table. A table extension in AL language may contain:

- ✓ Properties
- ✓ Fields
- ✓ Keys
- ✓ Global variables
- ✓ Table extension triggers

TABLE EXTENSION PROPERTIES

At this moment the table extension properties will not be described. Table extension allows for changing the standard properties. However not all properties can be changed. You can click **Ctrl+Space** in the object to check which properties can be changed.

TABLE EXTENSION FIELDS

You can create fields in the table extension if you need to add something to the standard object. You are doing it the same way as adding fields to the new table.

There is also possible to change the standard fields. However, only a few properties can be changed.

You cannot change the code which is triggered in the standard object, but you can add your code to the standard field in one of two triggers - **OnBeforeValidate()**, **OnAfterValidate()**. The code will be triggered either before or after the standard code.

HINTS

All fields that are added in the table extension must have a proper number – start with the same number as in your app.json range. Also, all fields need to have the affix.

Some of the tables work in Business Central "together" which means that they copy data between each other. Therefore, when adding an extension to one table, you need to think of adding a field not only to one table but to others as well. An example of such a case could be **Sales Header** – the field values are copied to **Sales Invoice Header** and **Sales Shipment Header**, also to **Sales Header Archive**.

TABLE EXTENSION KEYS

You can add new keys to the table extension. But you cannot mix standard fields with your new fields. You can add the keys which contain only added fields.

TABLE EXTENSION TRIGGERS

Like code for standard fields, you cannot change code for standard triggers that are in the table. However, you can add code that will be run before or after the standard code. For example to add code that will be run when deleting the record you can use **OnBeforeDelete()** or **OnAfterDelete()** triggers.

PAGE EXTENSION OVERVIEW

A page extension is an object which allows you to modify the standard page. A page extension in AL language may contain:

- ✓ Properties
- ✓ Controls
- ✓ Actions
- ✓ Global variables
- ✓ Page extension triggers

PAGE EXTENSION PROPERTIES

At this moment the page extension properties will not be described. Page extension allows for changing the standard properties. However not all properties can be changed. You can click **Ctrl+Space** in the object to check which properties can be changed.

PAGE EXTENSION CONTROLS

You can add new fields, created with table extension or standard fields, that are not present yet on the page. Adding the controls is very similar to adding controls to the standard page. However, you need to specify in which place the control should be added.

There is also possible to change the standard controls. However, only a few properties can be changed. You cannot change the code which is triggered in the standard object, but you can add your code to the standard control. The code will be triggered either before or after the standard code.

HINTS

All controls that are added need to have the affix.

Try to add the controls as last or first in a group – avoid putting them between existing controls.

Page extension actions

Similar to the controls, in the page extension, you can create new actions or modify existing ones. When you modify the standard action you can add the code which will be triggered after or before running the action.

Hints

All actions that are added need to have the affix.

Try to add the actions as last or first in a group – avoid putting them between existing controls.

Page extension triggers

In the page extension, you cannot change the code for standard triggers that are on the page. However, you can add code that will be run before or after the standard code.

Flow Fields – a special class of fields

The **Flow Fields** are a special class of fields. You can set them with the property **FieldClass**. For those fields, you cannot set **DataClassification**.

It allows to do easy mathematic operations, such **as count, sum, or get maximal** or **minimal value**. It also allows you to **show value from a different table** (lookup) or **check if the record exists**. To tell what the Flow Field should show, you can use the property **CalcFormula**.

For the Flow Fields, you should always set the editable property to false.

An example of how you can create a CalcField formula you can see when you hover over its name.

```
(property) CalcFormula: String
Sets the Calculation formula for a FlowField. The following syntax is valid for the CalcFormula property:
CalcFormula =
[-]Exist(<DestinationTable> [WHERE (<TableFilters>)]) |
Count(<DestinationTable> [WHERE (<TableFilters>)]) |
[-]Sum(<DestinationTable>.<DestinationFieldName> [WHERE(<TableFilters>)])|
[-]Average(<DestinationTable>.<DestinationFieldName> [WHERE(<TableFilters>)]) |
Min(<DestinationTable>.<DestinationFieldName> [WHERE(<TableFilters>)]) |
Max(<DestinationTable>.<DestinationFieldName> [WHERE(<TableFilters>)]) |
Lookup(<DestinationTable>.<DestinationFieldName> [WHERE(<TableFilters>)])
<TableFilters> ::= <TableFilter> {,<TableFilter>}
<TableFilter> ::=
```

TASK: SHOW THE NUMBER OF BONUSES FOR CUSTOMER

The system architect just checked what you did so far and is happy. But there are new requirements:

- ✓ On the Customer List page, the user should be able to see the number of bonuses assigned to the customer
- ✓ A user should be able to navigate to a list of bonuses assigned to the customer

1. In the new folder **Customer**, create a new file **Customer.TableExt.al** and create a new table extension **MNB Customer** using snippet **ttableext** (double "t"). Make sure that it extends the Customer table
2. Add a new field **MNB Bonuses** which should be a type **Integer**
3. Change the field class to **FlowField** and add **CalcFormula**. It should count records from the **MNB Bonus Header** table where the field **Customer No.** is the same as **No.** from the **Customer** table
4. Make sure the field is not editable and has a proper caption (without prefix)

SOLUTION

```
tableextension 65400 "MNB Customer" extends Customer
{

    fields
    {
        field(65400; "MNB Bonuses"; Integer)
        {
```

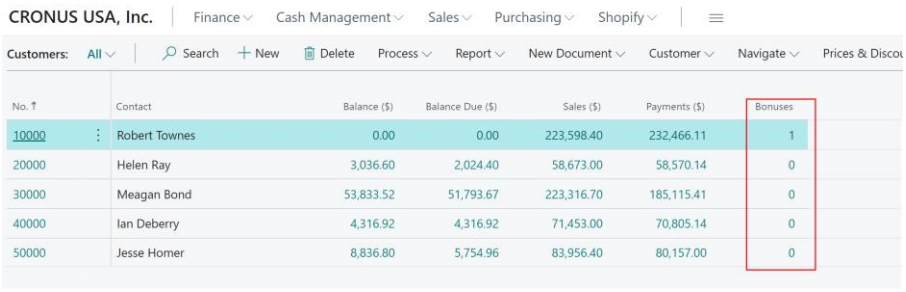
```
            Caption = 'Bonuses';
            FieldClass = FlowField;
            CalcFormula = count("MNB Bonus Header" where("Customer No." =
field("No.")));
            Editable = false;
        }
    }
}
```

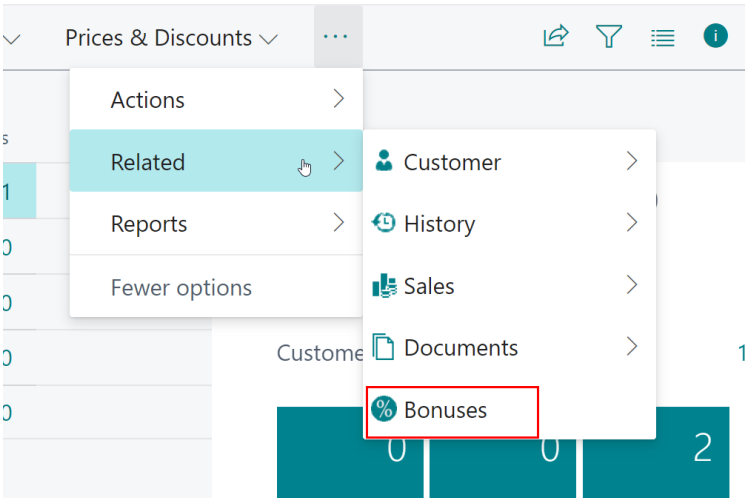
5. In the same folder create a new file **CustomerList.PageExt.al** and create a new page extension **MNB Customer List** using snippet **tpageext**. Make sure that it extends the **Customer** table
6. Add new control **MNB Bonuses** at the end of **Control1**.
7. Remember to add **ToolTip** and **ApplicationArea** properties
8. Create new action **MNBBonuses**. Action should open the **Bonus List** page where **Customer No.** is the same as field **No.** in the **Customer** table
9. Remember to add proper properties to the action
10. Publish your code to see the changes

SOLUTION

```
pageextension 65400 "MNB Customer List" extends "Customer List"
{
    layout
    {
        addlast(Control1)
        {
            field("MNB No. of Bonuses"; Rec."MNB Bonuses")
            {
                ApplicationArea = All;
                ToolTip = 'Specifies the number of assigned bonuses to
the customer.';
            }
        }
    }
    actions
    {
        addlast(navigation)
        {
```

```
            action(MNBBonuses)
            {
                Caption = 'Bonuses';
                ToolTip = 'Open the list of bonuses assigned to the
customer.';
                ApplicationArea = All;
                Image = Discount;
                RunObject = page "MNB Bonus List";
                RunPageLink = "Customer No." = field("No.");
            }
        }
    }
}
```





CHAPTER SUMMARY

- ✓ In this chapter, you understood how to develop pages and table extensions.
- ✓ You developed the first table and page extension.

CHAPTER 6

BASIC AL STATEMENTS AND METHODS

OBJECTIVES

In this chapter, you will get information on how to add simple logic to the fields and tables.

The objectives are:

- ✓ Get familiar with procedures and variables
- ✓ Get familiar with basic statements
- ✓ Understand how to retrieve records from the database using Get() and using Filters
- ✓ Understand how to show messages to the user

PROCEDURES

In AL you can write code directly in the table, fields, pages, or action triggers. For example **OnInsert()** in table Bonus Header, **onValidate()** in field Starting Date in that table, **onAction()** in Customer Card on Page Bonus Card.

However, the code in such places cannot be reused and also impact the readability of the overall solution. Therefore, often the code can be placed in the procedures. You can find basic types of the procedures:

- ✓ **Global** – you will be able to run the procedure from any extension and any other object
- ✓ **Local** – you will be able to run procedure only in the same object
- ✓ **Internal** – you will be able to run the procedure from any object but not from different extensions

Each procedure needs to have a name and **begin** and **end**. Do not use spaces in the names.

```
local procedure MyLocalProcedure()
begin

end;

procedure MyGlobalProcedure()
begin

end;

procedure MyInternalProcedure()
begin

end;
```

HINTS

Always use meaningful names for the procedure so everyone can understand right away what such a procedure does.

Using the context menu, you can find where the procedure is used or go directly to the definition of this procedure

Each procedure can have parameters that you can define in the procedure header. If you want to modify a parameter in the function, you can pass it by reference.

```
    local procedure MyLocalProcedure(Customer: Record Customer)
    begin

    end;

    procedure MyGlobalProcedure(Customer: Record Customer; var
SalesHeader: Record "Sales Header")
    begin

    end;
```

The above example shows a parameter to **MyLocalProcedure** called Customer and it passes the Customer record. To **MyGlobalProcedure** are passed two record parameters – **Customer** and **SalesHeader** – the second one will be changed inside the procedure and the changed value will be returned.

HINTS

It is possible to overload the procedure which means there can exist the same procedures in the same object but with different parameters.

The procedure can also return some value. For that, you need to specify the type of variable which is returned. The return value you can specify in the brackets of method exit().

```
local procedure MyLocalProcedure(): Integer
begin
    exit(10);
end;
```

The above example returns the Integer value 10.

VARIABLES

In AL you can add variables in the code. At this moment you need to know that there are two basic kinds of variables:

- ✓ **Global** – the variable is used in the whole instance of the object
- ✓ **Local** – the variable is used only in the single run of the procedure or trigger

Each variable has its own type. Many of the types are the same as field types (such as Integer, Code, Text, Enum, Date, DateTime, etc.). However, some other types are good to know.

Label	This type is used to store text constants for example to show questions, messages, and errors.
Record	**The record** type is used when the record needs to be retrieved from the database.
Page	Page type can be used to open a specified page.
Report	**Report** type can be used to open a specified page.

To declare the variable, you need to decide if it is local or global and placed it in the proper area – either in the object or in the function. The below example shows how to specify the global and local variables.

```
var
    SalesHeader: Record "Sales Header";
    NoOfCustomers, NoOfSalesOrders: Integer;

local procedure MyLocalProcedure()
var
```

```
    Customer: Record Customer
begin

end;
```

To assign value to the variable you can use **:=**.

```
var
    NoOfCustomers: Integer;

local procedure MyLocalProcedure()
begin
 NoOfCustomers := 1;
end;
```

HINTS

*Keep global variables in one place – do not create many sections of **var**.*

If you have variables of the same type, you can add them in the same line (see above Integers)

You do not need to add an affix to the variable names.

*When creating variables always use meaningful names. When creating object variables such as Record, Page, Report, etc. use the same name as the name of the object but without affix and space for example **Record "Sales Header"** name **SalesHeader** (not SalesOrder).*

Try to keep variables in order.

REC – SPECIAL VARIABLE ON PAGE AND TABLES

If you want to use in the object such as Table, Table Extension, Page, or Page Extension current value of the record you do not need to declare it as a separate variable.

To retrieve the value of the current record you can use **Rec**.

```
local procedure MyLocalProcedure()
begin
    if Rec."No." = '10000' then begin
      …
    end;
end;
```

The above example checks if the **current record** has 10000 in the field "No.".

HINTS

Sometimes the Rec is omitted in the code – it means that the variable is related to the current record.

IF … ELSE STATEMENT

The **if...else** statement is very common in any programming language. It allows you to run some code only if some conditions or conditions are true. With the else part of the statement, you can tell what should happen if the conditions are not met. Examples you can find below.

```
    if a > b then begin
      …
    end;

    if (a > b) and not (c < b) then
        Message('Hello World');

    if (a > b) and not (c < b) then
        Message('Hello World')
    else begin
      …
    end;
```

HINTS

*The **else** part is not always needed.*

You can use it and/or if there are more conditions.

*If some condition should not be true, then you can use **not** before it.*

*If more than one line should be executed after the if statement, then put lines between **begin** and **end**. If there is only one line, then do not use **begin** and **end**.*

GET RECORDS BASED ON PRIMARY KEY

Method **Get()** is used to get the record values from the database by using the **primary key fields**. In the parameters, you should add values of the fields in the same order as in the defined key. Using this method will always return you only one record.

```
var
  Customer: Record Customer;
  SalesHeader: Record "Sales Header";
begin
 Customer.Get('10000');
end;
```

Above example Get the values of customers that have the number 10000.

HINTS

In some places, you would see code that does not have any value in Get(). It means that the record that will be retrieved has in the Primary Key one field and the value of such field is empty. This is valid for all Setup tables such as General Ledger Setup, Sales & Receivable Setup, Inventory Setup, etc.

If in the primary key, you would have more than one field add the fields with come. For example, to get the record from the table Job Task you need to specify Job No. and Job Task No. in the Get – JobTask.Get('J0010','JT9999');

The above examples are only to show you how you need to get the data. However, never hardcode the values in the code.

GET RECORDS BASED ON FILTER

Often it is needed to get the record or set of records based on fields in the table. For example, to filter all customers that are from a particular country, or filter all Sales Orders for one customer.

For that, you can use the below methods.

SETRANGE()

The **SetRange()** method allows you to add a simple filter. You can either put the range of values that you want to filter, or you can put only one value. It means that the filter will be set to one value.

```
var
  Customer: Record Customer;
  SalesHeader: Record "Sales Header";
begin
  Customer.Get('10000');
  SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
  SalesHeader.SetRange("Document Type",Enum::"Sales Document
Type"::Order);
end;
```

The above example filters the Sales Header table for a specific customer where the document type is Order.

SETFILTER()

The **SetFilter()** method allows you to add a more complex filter. You can put in filter any combination of the operators such as <, >, .. , &, |, and =. In the filter string, you can use also value replacements like %1, %2, and so on.

```
var
  Customer: Record Customer;
  SalesHeader: Record "Sales Header";
begin
  Customer.Get('10000');
```

```
  SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
  SalesHeader.SetFilter("Document Type",'%1|%2',Enum::"Sales Document
Type"::Order,Enum::"Sales Document Type"::Invoice);

end;
```

The above example filters the Sales Header table for a specific customer where the document type is Order **or** Invoice.

HINTS

*Neither **SetFilter()** nor **SetRange()** by itself does not give values for the record – it only set the filters. To get values you need to use other methods described below.*

RESET()

The **Reset()** method is used to remove all filters which are applied to the record variable. Thanks to that you can be sure that the set of records that you will get is not filtered in any way.

```
var
  Customer: Record Customer;
  SalesHeader: Record "Sales Header";
begin
  Customer.Get('10000');
  SalesHeader.Reset();
  SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
  SalesHeader.SetFilter("Document Type",'%1|%2',Enum::"Sales Document
Type"::Order,Enum::"Sales Document Type"::Invoice);

end;
```

The above example removes any filters applied to the Sales Header that could be potentially added in previous lines of code.

HINTS

You do not need to reset the filters if you are sure that the filters have not been applied before. For example, local variables when first-time users do not require a reset of filters.

If you want to change the filter on the same variable for a specific field, you do not need to reset it. You can simply apply the next value to the same field filter.

FINDFIRST(), FINDLAST()

You can retrieve the first or last record after filtering records in the database using one of the methods: **FindFirst()** or **FindLast()**.

```
var
  Customer: Record Customer;
  SalesHeader: Record "Sales Header";
begin
  Customer.Get('10000');
  SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
  SalesHeader.SetFilter("Document Type",'%1|%2',Enum::"Sales Document
Type"::Order,Enum::"Sales Document Type"::Invoice);
  SalesHeader.FindFirst()
end;
```

The above example filters the Sales Header table for a specific customer where the document type is Order or Invoice **and gets values of the first record that meets conditions**.

HINTS

*Methods such as **FindFirst** or **FindLast** are used if you need to get **only one** record based on fields that are not part of the primary key (or you do not know all values).*

*If the record will not be found, you will see the error. This is why very often before FindFirst or FindLast you would see **if… then**.*

FINDSET()

You can retrieve the set of records within the filters using the **FindSet()** statement. Later you can use the statement **Repeat..Until** to move between records.

```
var
 Customer: Record Customer;
 SalesHeader: Record "Sales Header";
begin
 Customer.Get('10000');
 SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
 SalesHeader.SetFilter("Document Type",'%1|%2',Enum::"Sales Document
Type"::Order,Enum::"Sales Document Type"::Invoice);
 SalesHeader.FindSet();
end;
```

The above example filters the Sales Header table for a specific customer where the document type is Order or Invoice **and gets values of the set of records that meets conditions**.

HINTS

FindSet *is used if you need to get a set of records this is why very often before it you can find* ***SetRange*** *or* ***SetFilter****.*

If the record will not be found, you will see the error. This is why very often before FindFirst or FindLast you would see ***if… then****.*

If you would not get retrieve records in the loop, then you will see values from the first record. This is why the FindSet is almost always used with ***Repeat…Until*** *loop.*

REPEAT … UNTIL

A **repeat...until** statement gives you the possibility to run the same code in the loop until the statement which you want will not be true.

Typically, the statement is used to navigate thru the records. An example of it you can find below. You can filter the records before the statement to get in the loop only records that you would like to process. To check if the record is the last one in the set you can use **Next() = 0**.

```
var
 Customer: Record Customer;
 SalesHeader: Record "Sales Header";
begin
 Customer.Get('10000');
 SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
 SalesHeader.SetFilter("Document Type",'%1|%2',Enum::"Sales Document
Type"::Order,Enum::"Sales Document Type"::Invoice);
 SalesHeader.FindSet();
 repeat
   …
 until SalesHeader.Next() = 0;

end;
```

The above example filters the Sales Header table for a specific customer where document type is Order or Invoice **and repeats operation between repeat … until for each record found in the set**.

COUNT()

The **Count()** statement allows you to count records in the applied filters.

```
var
  Customer: Record Customer;
  SalesHeader: Record "Sales Header";
  NoOfRecords: Integer;
begin
  Customer.Get('10000');
  SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
  SalesHeader.SetRange("Document Type",Enum::"Sales Document
Type"::Order);
  NoOfRecords := SalesHeader.Count;
end;
```

The above example filters the Sales Header table for a specific customer where document type is Order and **assigns to integer value number of such records**.

IsEmpty()

The **isEmpty()** statement allows you to check if the filter would return zero records.

```
var
  Customer: Record Customer;
  SalesHeader: Record "Sales Header";
  NoOfRecords: Integer;
begin
  Customer.Get('10000');
  SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
  SalesHeader.SetRange("Document Type",Enum::"Sales Document
Type"::Order);
  if SalesHeader.IsEmpty then
    begin
      …
    end;

end;
```

The above example filters the Sales Header table for a specific customer where the document type is Order and **checks if the set of records would be empty (would not be any record in that set)**.

```
var
  Customer: Record Customer;
  SalesHeader: Record "Sales Header";
  NoOfRecords: Integer;
begin
  Customer.Get('10000');
  SalesHeader.SetRange("Sell-to Customer No.",Customer."No.");
  SalesHeader.SetRange("Document Type",Enum::"Sales Document
Type"::Order);
  if not SalesHeader.IsEmpty then
    begin
      …
    end;
```

```
end;
```

The above example filters the Sales Header table for a specific customer where the document type is Order and **checks if the set of records would be not empty (would find any records in the set)**.

Hints

if not isEmpty then *does not return values in the record. It means if the record would exist you will not know what are the values of it. In other words, it returns values of yes or no.*

Show Messages and Errors

If you can show a message to the user, you can use the method **Message()**. You can also show an error on the screen. For that, you can use the method **Error()**. When you use the **Error()** method then the transaction will be stopped and rollback.

In both methods, you can use the word replacement as %1, %2, etc. When you do so add the comment what %1 or %2 means.

You never should hardcode the message shown on the screen. All text values should be stored in the variables in the object. You should create a **Label** variable and assign the message or error text to it.

Examples you can find below.

```
var
 Customer: Record Customer;
 CustomerNameMsg: Label 'Customer Name is %1.', Comment = '%1 - customer
name';
begin
 Customer.Get('10000');
 Message(CustomerNameMsg,Customer.Name);
end;
```

The above example shows a **message** with the customer's name.

```
var
 Customer: Record Customer;
 CustomerNameErr: Label 'Customer Name is %1.', Comment = '%1 - customer
name';
begin
 Customer.Get('10000');
 Error(CustomerNameMsg,Customer.Name);
end;
```

The above example shows an **error** with the customer name.

HINTS

Variables with messages and errors can be stored locally or globally.

Never hardcode the text in the code. It would have an impact on translations and other features such as telemetry.

*Add proper suffix to the variable name. For messages finish the name with **Msg**, for errors finish the name with **Err**.*

TASK: BLOCK DELETING THE CUSTOMER WHEN AT LEAST ONE BONUS EXISTS

It turns out that you can delete the customer when any bonus exists for the customer. Your system architect would like to block it.

1. Open file **Customer.TableExt.al** and add new trigger **OnBeforeDelete()**
2. Create a new local procedure **TestIfBonusExists**

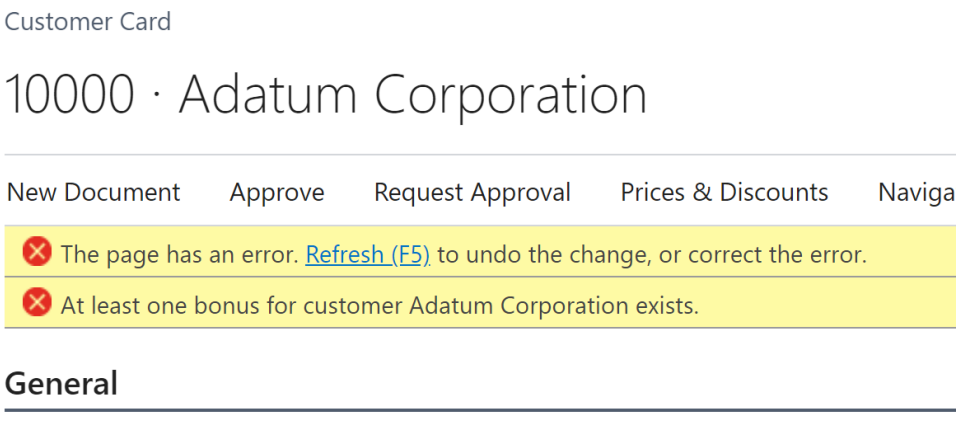
HINTS

*To create a procedure, you can use the snippet **procedure**.*

3. Filter Bonus Header Table where the customer is used and show an error message if any record exists
4. Add the procedure to the created trigger
5. Publish your code, create a Bonus for the customer and try to delete the customer after

SOLUTION

```
trigger OnBeforeDelete()
begin
  TestIfBonusExists();
end;

var
  AtLeastOneBonusForCustomerExistsErr: Label 'At least one bonus for
customer %1 exists.', Comment = '%1 - customer name';

local procedure TestIfBonusExists()
var
 BonusHeader: Record "MNB Bonus Header";
begin
 BonusHeader.SetRange("Customer No.", Rec."No.");
 if not BonusHeader.IsEmpty then
   Error(AtLeastOneBonusForCustomerExistsErr, Rec.Name);
 end;
```



TASK: CHECK IF STARTING AND ENDING DATES ARE IN THE CORRECT ORDER

During testing, it turns out that when creating the Bonus Card Starting Date can be later than the Ending Date. This is not correct.

- ✓ Make sure that the Ending date is automatically filled when is before starting date (also if is empty)
- ✓ Make sure that Starting Date and Ending Date are in the correct order.

1. Open file **BonusHeader.Table.al** and add new triggers **OnValidate()** for Starting and Ending Dates
2. Add the code which will check if the Starting Date or Ending Date is in the correct order and if not replace the values. For example, if **Ending Date** is before **Starting Date** then **Starting Date** is changed to the same value as **the Ending Date**

SOLUTION

```
field(3; "Starting Date"; Date)
        {
            DataClassification = CustomerContent;
            Caption = 'Starting Date';
            trigger OnValidate()
            begin
                if "Ending Date" < "Starting Date" then
                    "Ending Date" := "Starting Date";
            end;
        }
field(4; "Ending Date"; Date)
        {
            DataClassification = CustomerContent;
            Caption = 'Ending Date';
            trigger OnValidate()
            begin
                if "Ending Date" < "Starting Date" then
                    "Starting Date" := "Ending Date";
            end;
        }
```

TASK: BLOCK CHANGING THE HEADER IF THE STATUS IS RELEASED

The system architect just found out that you can modify the header information when the status is released.

- ✓ It is needed to block modification of the fields: "Customer No.", "Starting Date", and "Ending Date".
- ✓ Block delete the Bonus when the status is Released.

1. Open file **BonusHeader.Table.al** and add new local function **TestStatus**
2. Check if the Status field is Released. If so, then show an error

   ### HINTS

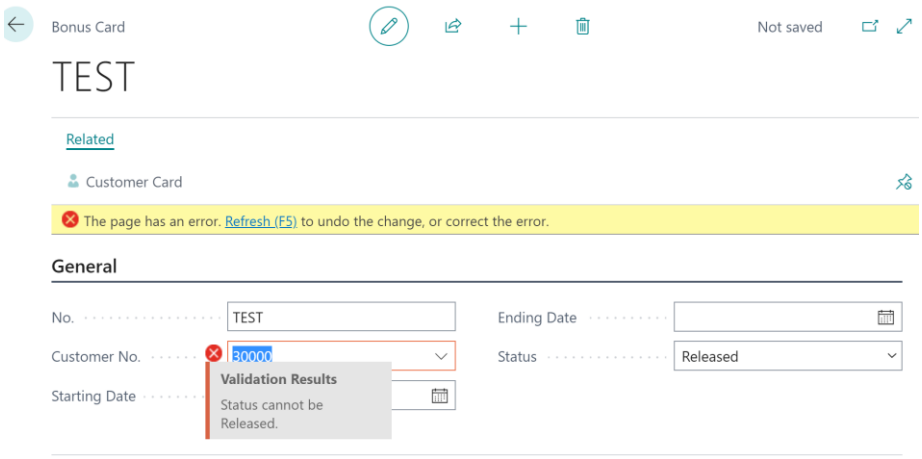
   Use if statement to see if Status is equal to Released. (to do that you can after Status field name :: (double ":") to get available values for the enum.

3. Add the function to all fields in trigger **OnValidate()** and in trigger **OnDelete()**

SOLUTION

```
field(2; "Customer No."; Code[20])
 {
   DataClassification = CustomerContent;
   Caption = 'Customer No.';
   TableRelation = Customer;
   trigger OnValidate()
   begin
     TestStatus();
   end;
 }

…

trigger OnDelete()
begin
  TestStatus();
end;
```

```
var
  StatusCannotBeReleasedErr: Label 'Status cannot be %1.', Comment = '%1
status field value';

local procedure TestStatus()
begin
 if Status = Status::Released then
   Error(StatusCannotBeReleasedErr, Status);
end;
```



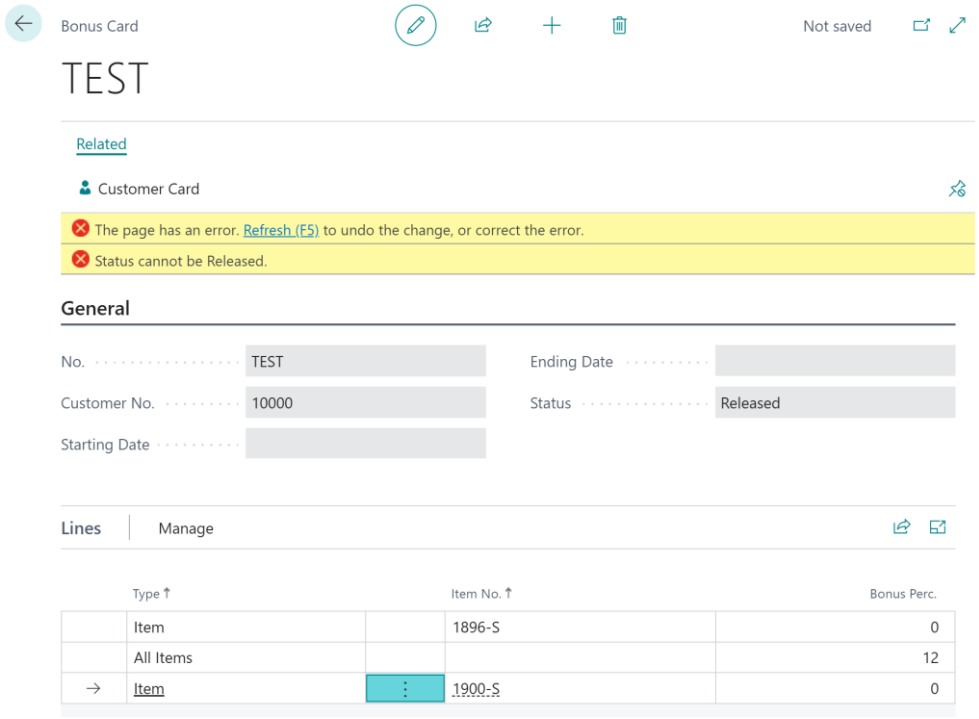
Task: Block changing the Lines if the status is Released

Great now it is not possible to change the header but it is still not perfect. What about the lines? Now you need to block create, modify or delete the lines for the Bonus if the status is released

1. Open file **BonusLine.Table.al** and add new local function **TestStatus**
2. Check if the **Header Status** field is Released. If so, then show an error
3. Add the function to all fields in trigger in triggers **OnDelete(), OnInsert(), OnModify(), onRename()**

SOLUTION

```
trigger OnInsert()
    begin
        TestStatus();
    end;

    trigger OnModify()
    begin
        TestStatus();
    end;

    trigger OnDelete()
    begin
        TestStatus();
    end;

    trigger OnRename()
    begin
        TestStatus();
    end;

    var
        StatusCannotBeReleasedErr: Label 'Status cannot be %1.', Comment
= '%1 status field value';

    local procedure TestStatus()
    var
        BonusHeader: Record "MNB Bonus Header";
    begin
        if BonusHeader.Get(Rec."Document No.") then
            if BonusHeader.Status = BonusHeader.Status::Released then
                Error(StatusCannotBeReleasedErr, BonusHeader.Status);
    end;
```



OPERATIONS ON RECORDS

In AL language it is possible to do the operations on the records such as insert, modify, and delete. It is also possible to do bulk operations on the set of records that have been filtered before.

ASSIGN A VALUE TO THE TABLE FIELD

There are two basic methods to assign value to the table field. You can assign a value directly or you can force the system to validate the field. If you would choose to validate the field it means that the code is in the table in trigger **OnValidate()** for the field will be executed. Examples of methods you can find below.

```
Customer."Credit Limit (LCY)" := 0;

Customer.Validate("Credit Limit (LCY)" , 0);
```

Hints

If you assign the value to the fields in the tables which are posted documents or ledger entries, in most cases, you can assign value without validating it.

When you assign the value to the field remember that the value would not be saved in the database until you will not insert or modify the record.

Init() and Insert()

The **Init()** method you will use when you want to start writing to the database a new record.

Normally, after **Init()** and assigning values to the fields, you will use the method **Insert()**. By specifying the parameter in method **Insert()**, you can decide if code from **OnInsert()** trigger will be run. If you will leave brackets empty, then it means, that code will not be triggered.

```
var
  Customer: Record Customer;
  MyName, MyAddress : Text[100];
begin
  Customer.Init();
  Customer.Name := MyName;
  Customer.Address := MyAddress;
  Customer.Insert();
end;
```

In the above example insert the new Customer **without triggering the code** which is in **OnInsert** in the Customer Table.

```
var
  Customer: Record Customer;
  MyName, MyAddress : Text[100];
begin
  Customer.Init();
  Customer.Name := MyName;
  Customer.Address := MyAddress;
```

```
  Customer.Insert(true);
end;
```

HINTS

Remember that in the database can be only one record with the same primary key. It means if you would try to add a record that already exists in the database, you will see the error.

MODIFY() AND MODIFYALL()

Method **Modify()** allows you to modify the record which you retrieved from the database. It means that first one of the methods such as **FindFirst()**, **FindLast()**, **FindSet()** or **Get()**. By specifying the parameter in method **Modify()**, you can decide if code from **OnModify()** trigger will be run. If you will leave brackets empty, then it means, that code will not be triggered.

```
var
  Customer: Record Customer;
  MyName, MyAddress : Text[100];
begin
  Customer.Get('10000');
  Customer.Name := MyName;
  Customer.Address := MyAddress;
  Customer.Modify(true);
end;
```

The above example gets the Customer with the number 10000, modify the Name and Address fields, **and trigger** which is in **OnModify** in the Customer Table.

HINTS

If the record does not exist in the database, you cannot modify it. It will give you the error.

If you inserted a record few lines of code above, you do not need to get it to do the modification. In other words, after inserting the record you can assign a new value to the fields and then modify the record (without getting it again).

Method **ModifyAll()** allows you to modify **all records** that are within the filter. It means that before modifying all records you should use one of the methods such as **SetFilter()** or **SetRange()** to get filtered records. You do not need to retrieve them from the database – the filter is enough.

ModifyAll() has some mandatory parameters. You need to specify the field which will be modified in the first parameter and in the second parameter value of that field. In the third parameter, you can decide if code from **OnModify()** trigger will be run.

```
var
 Customer: Record Customer;
 MyPaymentCode, MyCountryCode : Code[10];
begin
 Customer.SetRange("Country/Region Code", MyCountryCode);
 Customer.ModifyAll("Payment Method Code", MyPaymentCode, true);
end;
```

The above example filters all Customers for a specific Country and modifies all records with a new Payment Method. It also runs the trigger **onModify()** of the Customer table.

HINTS

*Using **ModifyAll()** remember to add a filter before. Otherwise, you will modify all records in the table.*

DELETE() AND DELETEALL()

Method **Delete()** allows you to delete records that you retrieved from the database. It means that first one of the methods such as **FindFirst()**, **FindLast()**, **FindSet()** or **Get()**. By specifying the parameter in method **Delete()**, you can decide if code from **OnDelete()** trigger will be run. If you will leave brackets empty, then it means, that code will not be triggered.

```
var
  Customer: Record Customer;
begin
  Customer.Get('10000');
  Customer.Delete(true);
end;
```

The above example gets the Customer with the number 10000, deletes it, **and triggers** which are in **OnDelete** in the Customer Table.

HINTS

If the record does not exist in the database, you cannot delete it. It will give you the error. The same will happen if you will not retrieve the record first from the table.

Method **DeleteAll()** allows you to delete **all records** that are within the filter. It means that before deleting all records you should use one of the methods such as **SetFilter()** or **SetRange()** to get filtered records. You do not need to retrieve them from the database – the filter is enough.

By specifying the parameter in method **DeleteAll()**, you can decide if code from **OnDelete()** trigger will be run. If you will leave brackets empty, then it means, that code will not be triggered.

```
var
```

```
 Customer: Record Customer;
 MyCountryCode : Code[10];
begin
 Customer.SetRange("Country/Region Code", MyCountryCode);
 Customer.DeleteAll();
end;
```

Above example filter all Customers for a specific Country and delete all. It also runs the trigger **onDelete()** of the Customer table.

HINTS

*Using **DeleteAll()** remember to add a filter before. Otherwise, you will delete all records in the table.*

If there will be no records in the filter you will not get an error and no records will be deleted.

TASK: REMOVE BONUS LINES WHEN THE BONUS IS DELETED

One of the testers found out that when deleting the Bonus Header the lines are not deleted. It is not good because leaves no needed records in the database. You need to remove them automatically.

1. Open file **BonusHeader.Table.al** and add new local function **DeleteLines**
2. Add code that deletes all lines which have the same **Document No.** as **No.**
3. Add the function to the trigger **OnDelete()**. Remember about the order when the delete should be performed – it should be after checking **Status**

SOLUTION

```
trigger OnDelete()
begin
    TestStatus();
    DeleteLines();
end;
```

```
local procedure DeleteLines()
var
    BonusLine: Record "MNB Bonus Line";
begin
    BonusLine.SetRange("Document No.", "No.");
    BonusLine.DeleteAll();
end;
```

CHAPTER SUMMARY

- ✓ In this chapter, you get familiar with how to get, modify, delete and filter records
- ✓ You added the first logic to Bonus Extension
- ✓ You also learned how to show messages and errors.

CHAPTER 7

CODEUNITS AND EVENTS

OBJECTIVES

In this chapter, how to create the codeunits and also how to extend the logic of standard Business Central

The objectives are:

- ✓ Get familiar with codeunits
- ✓ Understand how to create event subscribers
- ✓ Understand how to create events in your solution
- ✓ Develop a table and page to gather the bonus entries
- ✓ Develop code to calculate the bonus from sales invoices

CODEUNIT OVERVIEW

A codeunit is an object which allows you to write a code that will be executed. In a codeunit, you have one standard trigger called **onRun()**. This trigger, as the name says, is triggered when you run the codeunit.

In a codeunit, you can create procedures that have been described in the previous chapter. One of the special kinds of the procedure is even subscribers.

EVENT PUBLISHERS AND SUBSCRIBERS

In the AL language, you cannot modify the standard code of the application. Sometimes, you will see that you need to add some code when some standard code is triggered. For example, if someone posts a sales invoice you want to calculate the bonus.

To do so, you need to subscribe to an event that is published by Microsoft in the base application extension or another extension provider (**this event is called a publisher**). The publishers always contain some parameters which you can use. An example of using the publisher in standard code you can find below. One publisher, to which you can subscribe, is just before (**OnBeforeSalesInvLineInsert**) and the second one is after (**OnAfterSalesInvLineInsert**) insert the **Sales Invoice Line** when posting sales documents.

```
900                 IsHandled := false;
901                 OnBeforeSalesInvLineInsert(SalesInvLine, SalesInvHeader, xSalesLine, SuppressCommit, IsHandled,
902                 if IsHandled then
903                     exit;
904                 if not IsNullGuid(xSalesLine.SystemId) then begin
905                     SearchSalesInvLine.SetRange(SystemId, xSalesLine.SystemId);
906                     if SearchSalesInvLine.IsEmpty() then begin
907                         SalesInvLine.SystemId := xSalesLine.SystemId;
908                         SalesInvLine.Insert(true, true);
909                     end else begin
910                         Session.LogMessage('0000DD6', SameIdFoundLbl, Verbosity::Warning, DataClassification::Sy
911                         SalesInvLine.Insert(true);
912                     end;
913                 end else begin
914                     SalesInvLine.Insert(true);
915                     Session.LogMessage('0000DDC', EmptyIdFoundLbl, Verbosity::Warning, DataClassification::Syste
916                 end;
917                 OnAfterSalesInvLineInsert(
918                   SalesInvLine, SalesInvHeader, xSalesLine, ItemLedgShptEntryNo, WhseShip, WhseReceive, Suppress
919                   SalesHeader, TempItemChargeAssgntSales, TempWhseShptHeader, TempWhseRcptHeader, PreviewMode);
```

As you can see both events allow you to retrieve such data as sales invoice header and line.

HINTS

The event publishers have the same naming convention. In most cases starts with the words OnBefore or OnAfter and is then followed by information at which point it is a trigger (for example Insert Sales Invoice Line). This allows an easy way to know where the event is placed.

To subscribe to the publisher (**it is called event subscriber**), you can use snippet **teventsub**. Later you will need to specify the object in which the publisher is and put the publisher's name. In the procedure parameters, you do not need to specify all parameters but only those which you will use. Remember only that the name of the parameters should be the same as in the publisher.

HINTS

*To not make a mistake in the parameters you can click **Ctrl+Space** to choose the parameter.*

You can declare the event subscriber only in codeunits.

*For naming the event subscriber use the pattern: Run + name of the publisher. For example, for publisher **OnAfterSalesInvLineInsert** use name **RunOnAfterSalesInvLineInsert**.*

TASK: CREATE A BONUS ENTRY TABLE AND BONUS ENTRIES PAGE

To store what bonus has been calculated for the bonus card you need to prepare first the table and page where you will have all the entries. In the next points, you will write the code which will insert data into this table. The table and page should not be editable. Users should be able to open the entries from the bonus card and list.

1. Create a new file **BonusEntry.Table.al** and add a new table **MNB Bonus Entry**
2. Create fields
    a. **Entry No.** – Integer and add property AutoIncrement
    b. **Bonus No.** - Code[20] with table relation to **MNB Bonus Header**
    c. **Document No.** - Code[20] with table relation to **Sales Invoice Header**
    d. **Item No.** - Code[20] with table relation to **Item**

e. **Posting Date** - Date

f. **Bouns Amount** - Decimal

3. Make sure that all fields are not editable
4. Make sure that **Entry No.** is the primary key
5. Remember to add the table to the **MNB Bonus Reg.** permission set

SOLUTION

```
table 65402 "MNB Bonus Entry"
{
    DataClassification = CustomerContent;
    Caption = 'Bonus Entry';

    fields
    {
        field(1; "Entry No."; Integer)
        {
            DataClassification = CustomerContent;
            Caption = 'Entry No.';
            Editable = false;
            AutoIncrement = true;
        }
        field(2; "Bonus No."; Code[20])
        {
            DataClassification = CustomerContent;
            Caption = 'Bonus No.';
            Editable = false;
            TableRelation = "MNB Bonus Header";
        }
        field(3; "Document No."; Code[20])
        {
            DataClassification = CustomerContent;
            Caption = 'Document No.';
            Editable = false;
            TableRelation = "Sales Invoice Header";
        }
        field(4; "Item No."; Code[20])
        {
            DataClassification = CustomerContent;
            Caption = 'Item No.';
            Editable = false;
            TableRelation = Item;
        }
        field(5; "Posting Date"; Date)
        {
```

```
            DataClassification = CustomerContent;
            Caption = 'Posting Date';
            Editable = false;
        }
        field(6; "Bonus Amount"; Decimal)
        {
            DataClassification = CustomerContent;
            Caption = 'Bonus Amount';
            Editable = false;
        }
    }
    keys
    {
        key(PK; "Entry No.")
        {
            Clustered = true;
        }
    }
}
```

```
permissionset 65400 "MNB Bonus Reg."
{
    Caption = 'Bonus Registration';
    Assignable = true;
    Permissions =
        tabledata "MNB Bonus Header" = RMID,
        tabledata "MNB Bonus Line" = RMID,
        tabledata "MNB Bonus Entry" = RMID;
}
```

6. Create a new file **BonusEntries.Page.al** and add a new page **MNB Bonus Entries**. It should be based on the table **MNB Bonus Entry** and should be a type List
7. Make sure that it is not possible to edit the list and that the user cannot do any operation on it (insert, modify and delete)
8. Remember about mandatory properties for the page and fields. Add the page to the **UsageCategory** that is **History**

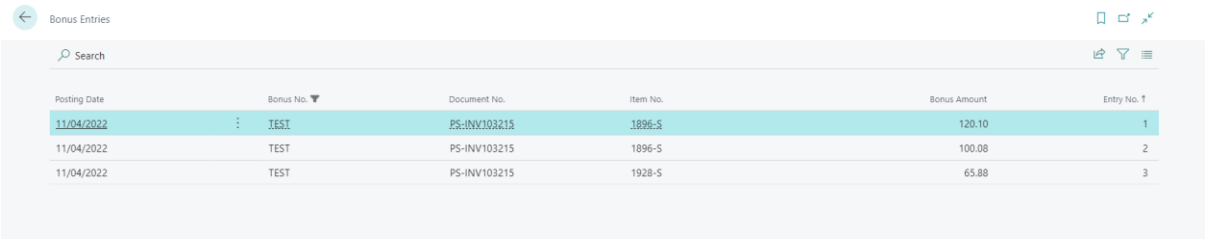
9. Add all fields from the table to the page. Make sure to add **Posting Date** as the first column and **Entry No.** as the last column
10. Add page to the **MNB Bonus Entry** table as the **LookupPageId** and **DrillDownPageId**

SOLUTION

```
page 65403 "MNB Bonus Entries"
{
    PageType = List;
    SourceTable = "MNB Bonus Entry";
    Editable = false;
    DeleteAllowed = false;
    InsertAllowed = false;
    ModifyAllowed = false;
    Caption = 'Bonus Entries';
    UsageCategory = History;
    ApplicationArea = All;

    layout
    {
        area(Content)
        {
            repeater(Control1)
            {
                field("Posting Date"; Rec."Posting Date")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies sales invoice posting date.';
                }
                field("Bonus No."; Rec."Bonus No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies bonus number.';
                }
                field("Document No."; Rec."Document No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies sales invoice number.';
                }
                field("Item No."; Rec."Item No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies item number.';
                }
                field("Bonus Amount"; Rec."Bonus Amount")
                {
```

```
                    ApplicationArea = All;
                    ToolTip = 'Specifies calculated bonus amount.';
                }
                field("Entry No."; Rec."Entry No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies entry number for the ledger.';
                }
            }
        }
    }
}
```

```
table 65402 "MNB Bonus Entry"
{
    DataClassification = CustomerContent;
    Caption = 'Bonus Entry';
    LookupPageId = "MNB Bonus Entries";
    DrillDownPageId = "MNB Bonus Entries";
```



HINTS

The reason to block editable on the table level and page level is to avoid the situation when other developers or companies that will extend the solution will not allow (even by mistake) to edit the records by creating a new page with the same source table.

11. Open the file **BonusCard.Page.al** and add new action **BonusEntries**
12. The action should open the page **MNB Bonus Entries** for the Bonus from which the action is run. Action should be placed in the area **Navigation**
13. Copy the action to the **MNB Bonus List** page

```
action(BonusEntries)
{
  ApplicationArea = All;
  Caption = 'Bonus Entries';
  Image = Entry;
  RunObject = page "MNB Bonus Entries";
  RunPageLink = "Bonus No." = field("No.");
  ToolTip = 'Open bonus entries.';
}
```

TASK: CREATE A CODEUNIT TO CALCULATE THE BONUS

The biggest task in front of you. You will create the codeunit which will calculate and insert the bonus entries. It will be run whenever the user will post the sales invoice for the item (only for the item). It needs to check if there are any bonuses for the customer in status Released within the date ranges defined in the Starting and Ending Date.

Remember that there might be more than one bonus assigned to the customer. You need to check if the bonus exists for all items and if exists for one single item.

If you would find the match, then you should insert the new record in the **MNB Bonus Entry** table. You will need to calculate the bonus as:

Sales Invoice Line **Line Amount** * Bonus Line **Bonus Percent** / 100.

1. Create a new file **BonusCalculation.Codeunit.al** and create a new codeunit. For that, you can use snippet **tcodeunit**. Codeunit should have the name **MNB Bonus Calculation**

   ### HINTS

   You can remove the code which is in the snippet related to trigger onRun(). It will not be needed

2. Create a new event subscriber. You can use snippet **teventsub**. Make sure to subscribe to the codeunit **"Sales-Post"** to publisher **OnAfterSalesInvLineInsert**

HINTS

Instead of writing manually the part 'OnSomeEvent' click Ctrl+Space to get available publishers. ElementName parameter set to empty (single quotes ' '), Parameters SkipOnMissingLicense and SkipOnMissingPermission set to true for now

3. Change the name of the procedure to **RunOnAfterSalesInvLineInsert** and add the only parameter - var SalesInvLine: Record "Sales Invoice Line"

HINTS

Instead of writing manually parameters click Ctrl+Space to get available.

4. Create a new local procedure **CalculateBonus** and add it to the event subscriber. Make sure to pass by reference (use var) to this procedure **Sales Invoice Line** record

HINTS

For clean code use the same name of the variable as in the event subscriber.

5. In the procedure, **CalculateBonus** check if **Sales Invoice Line** has a type different than **Item** - if yes then exit from the procedure

HINTS

Use the if statement to check if the Type field is equal to Item

6. In the procedure, **CalculateBonus** check if exists any bonus in status Released, for the same customer as **Bill-to Customer No.** in the sales invoice line. Check also if the **Posting Date** of the sales invoice line is between **Starting Date** and **Ending Date**

HINTS

Remember that there can be more than one bonus that will be in such a filter. Use SetRange and SetFilter to find records and then use Repeat…Until to process them.

7. If you find the bonuses for the assigned filter then check if in the bonus exists the line for all items - if yes then insert the record to table **MNB Bonus Entry**

HINTS

Add the code in a separate function and use the function inside Repeat…Until statement.

When inserting the record to the table "MNB Bonus Entry" set the "Entry No." field to 0. Because it is an auto incremental field it will automatically be set.

8. If you find the bonuses (remember that can be more than one) for the assigned filter then check if the bonus exists for the particular item - if yes then insert the record to table **MNB Bonus Entry**

HINTS

Add the code in a separate function and use the function inside Repeat…Until statement.

*You can create a new procedure to insert the bonus into the **MNB Bonus Entry t**able. Thanks to that you will have only one function to insert the data*

9. Publish your changes and test them
    a. Create Bonus in status released for correct dates and with some lines
    b. Create a Sale Invoice for the items and post it
    c. Check if the Entries are created correctly

SOLUTION

```
codeunit 65400 "MNB Bonus Calculation"
{
        [EventSubscriber(ObjectType::Codeunit,   Codeunit::"Sales-Post",
'OnAfterSalesInvLineInsert', '', true, true)]
    local procedure OnAfterSalesInvLineInsert(var SalesInvLine: Record
"Sales Invoice Line")
    begin
        CalculateBonus(SalesInvLine);
    end;

    local procedure CalculateBonus(var SalesInvLine: Record "Sales Invoice
Line")
    var
        BonusHeader: Record "MNB Bonus Header";
    begin
        if SalesInvLine.Type <> SalesInvLine.Type::Item then
            exit;
```

```
            BonusHeader.SetRange("Customer  No.",  SalesInvLine."Bill-to
Customer No.");
        BonusHeader.SetRange(Status, BonusHeader.Status::Released);
                    BonusHeader.SetFilter("Starting    Date",    '..%1',
SalesInvLine."Posting Date");
       BonusHeader.SetFilter("Ending Date", '%1..', SalesInvLine."Posting
Date");

        if BonusHeader.FindSet() then
            repeat
                FindBonusForAllItems(BonusHeader, SalesInvLine);
                FindBonusForOneItem(BonusHeader, SalesInvLine);
            until BonusHeader.Next() = 0;
    end;

     local procedure FindBonusForAllItems(var BonusHeader: Record "MNB
Bonus Header"; var SalesInvLine: Record "Sales Invoice Line")
    var
        BonusLine: Record "MNB Bonus Line";
    begin
        BonusLine.SetRange("Document No.", BonusHeader."No.");
        BonusLine.SetRange(Type, BonusLine.Type::"All Items");
        if BonusLine.FindFirst() then
            InsertBonusEntry(BonusLine, SalesInvLine);
    end;

    local procedure FindBonusForOneItem(var BonusHeader: Record "MNB Bonus
Header"; var SalesInvLine: Record "Sales Invoice Line");
    var
        BonusLine: Record "MNB Bonus Line";
    begin
        BonusLine.SetRange("Document No.", BonusHeader."No.");
        BonusLine.SetRange(Type, BonusLine.Type::Item);
        BonusLine.SetRange("Item No.", SalesInvLine."No.");
        if BonusLine.FindFirst() then
            InsertBonusEntry(BonusLine, SalesInvLine);
    end;

     local procedure InsertBonusEntry(var BonusLine: Record "MNB Bonus
Line"; var SalesInvLine: Record "Sales Invoice Line")
    var
        BonusEntry: Record "MNB Bonus Entry";
    begin
        BonusEntry.Init();
        BonusEntry."Entry No." := 0;
        BonusEntry."Bonus No." := BonusLine."Document No.";
        BonusEntry."Document No." := SalesInvLine."Document No.";
        BonusEntry."Item No." := SalesInvLine."No.";
```

```
        BonusEntry."Posting Date" := SalesInvLine."Posting Date";
            BonusEntry."Bonus Amount" := SalesInvLine."Line Amount" *
BonusLine."Bonus Perc." / 100;
        BonusEntry.Insert();
    end;
```

TASK: ADD EVENT PUBLISHERS

It is good to think about the extensibility of your solution. Your next task is to add two even publishers: one before inserting the **Bonus Entry** in the **InsertBonusEntry** procedure in the above solution, and one after inserting.

1. Open file **BonusCalculation.Codeunit.al** and create a new integration event at the bottom. Use snippet **teventint**. On the Integration Event level set both parameters in it to false (Include Sender and Access Global Variable)

   ### HINTS

   You may notice that there is no code inside the integration event.

2. Rename the function in the integration event to **OnBeforeInsertBonusEntry** and add two parameters: **Sales Invoice Line** and **Bonus Entry**. Remember to add var at least to Bonus Entry
3. Add the procedure before the line where the Bonus Entry is inserted.
4. Create a similar integration event for **OnAfterInsertBonusEntry** and add it after the record is inserted

SOLUTION

```
local procedure InsertBonusEntry(var BonusLine: Record "MNB Bonus Line";
var SalesInvLine: Record "Sales Invoice Line")
var
 BonusEntry: Record "MNB Bonus Entry";
 begin
   BonusEntry.Init();
   BonusEntry."Entry No." := 0;
```

```
    BonusEntry."Bonus No." := BonusLine."Document No.";
    BonusEntry."Document No." := SalesInvLine."Document No.";
    BonusEntry."Item No." := SalesInvLine."No.";
    BonusEntry."Posting Date" := SalesInvLine."Posting Date";
    BonusEntry."Bonus Amount" := SalesInvLine."Line Amount" *
BonusLine."Bonus Perc." / 100;
    OnBeforeInsertBonusEntry(BonusLine, SalesInvLine);
    BonusEntry.Insert();
    OnAfterInsertBonusEntry(BonusLine, SalesInvLine);
end;

[IntegrationEvent(false, false)]
local procedure OnBeforeInsertBonusEntry(var BonusLine: Record "MNB
Bonus Line"; var SalesInvLine: Record "Sales Invoice Line")
begin
end;

[IntegrationEvent(false, false)]
local procedure OnAfterInsertBonusEntry(var BonusLine: Record "MNB Bonus
Line"; var SalesInvLine: Record "Sales Invoice Line")
begin
end;
```

CHAPTER SUMMARY

- ✓ In this chapter, you understood how to develop codeunit and procedures.
- ✓ You know how to create an event subscriber to standard code.
- ✓ You know how to create your events.
- ✓ You developed an event that is run when the user posts the sales invoice. Now your bonuses are calculated and you can start thinking about improvements to the bonus extension.

CHAPTER 8

AUTOMATED TESTS

OBJECTIVES

In this chapter, you will get information on how to write simple automated tests for your extension.

Note that this chapter requires Docker Container at the moment of publishing this version.

The objectives are:

- ✓ Understand why you need to write automated tests
- ✓ Understand what the dependencies are
- ✓ Get familiar with the automated test structure
- ✓ Get familiar with the method TestField()
- ✓ Develop and run your first and simple automated test
- ✓ Find out where to get more information about the automated tests

AUTOMATE TESTS OVERVIEW

The automated tests are the tests that can be run multiple times to check if the code which has been written is matching the requirements. The advantage of such tests is that you write them once and then you can run them when it is needed. For example, check if your functionality works with the newest version which will be available for your customers after the next upgrade of Business Central.

Of course, the automated tests will not replace manual testing but can be a benefit for your process of writing the extensions.

There is no external tool for automated tests in Business Central. You can write them directly in the Visual Studio Code using the same language that you use for the extension development. However, Microsoft prepared useful libraries that allow you to write tests faster.

You can run the automated tests directly in Business Central. For that go to the **AL Test Tool** page and get test codeunits.

HINTS

When you created a Docker container one of the parameters was to install system test libraries. Test libraries are separate extensions that are installed in Business Central.

The examples shown in this chapter will be simple. If you would like to get more information about automated tests you can go and check the book ***Automated Testing in Microsoft Dynamics 365 Business Central by Luc van Vugt****.*

TEST APP

Although you can write tests in the same app as your modifications it is not a common approach. One of the reasons is that you do not need to install and run automated tests on customers' production environments.

This is why the tests are stored in a separate app.

Task: Create New App for the tests

You need to create a new app for tests. If you do not remember you can go back to Chapter 2. If you already have opened VSCode – open the first new window.

1. Create a new app **Bonus Registration Tests**
2. Open the **app.json** file and update properties such as **name**, **description, brief,** and **URL**
3. Create two folders on top-level **src** and **res**
4. Find your company logo (or any other file in jpg or png format) and copy it to the **res** folder
5. Update **app.json** file with logo path
6. Close your test app and open the app with the **Bonus Registration** extension.
7. Go to File and choose **Add Folder to Workspace**. Choose the test app. After that, you should see two apps in your Explorer

Dependencies in the App.json file

By default one extension does not see the symbols (objects) from the other extension. To be able to access the objects in different extensions it is needed to add dependencies in the **app.json** file.

Task: Add dependencies in the test app to the main app

In the test app, you need to add dependencies to the Bonus Registration app. For that, you will need to know information which you can find in the app.json file of Bonus Registration

1. Open the app.json file for the Test app and find property **dependency**
2. Inside the [] add a new dependency. Add {} and fill all properties. You can find the properties by clicking Ctrl+Space. You need to add the id, name, publisher, and version of the main (Bonus Registration) app.json file

   ### Hints

   The fastest way to do it without errors is simply to copy the four first rows from app.json of your Bonus Registration app.
3. Save the file and go to the Command Pallet (F1) and run function AL: Download Symbols

SOLUTION

```
"dependencies": [
  {
    "id": "4d35d9fd-22c9-452b-89c5-cf74cd47edda",
    "name": "Bonus Registration",
    "publisher": "myNAVblog.com",
    "version": "1.0.0.0"
  }
]
```

HINTS

Each app has a unique id so your solution will be different.

TEST CODEUNIT

To write the automated tests, you need to put them in the codeunits. You can have multiple tests in the same codeunit. This allows you to group tests for the same functionality.

It is required to set the codeunit property Subtype to be set as **Test**. Then the codeunit will be visible in the Business Central **AL Test Tool**.

Each test in the test codeunit is written as one procedure. You need to specify that the procedure is the Test. Only such procedures will be added to the **AL Test Tool** as test lines.

Example of the automated test with mark properties you can find below.

```
0 references
codeunit 50100 "MNB Bonus Card Tests"
{
    Subtype = Test;


    [Test]
    0 references
    procedure TestCustomerNoHasValueWhenRelease()
    begin
        // [SCENARIO] What is tested
        // [GIVEN] Given Some State
        // [WHEN] When Some Action
        // [THEN] Then Expected Output
    end;
}
```

TEST STRUCTURE

The automated test contains the below sections:

- [SCENARIO]
- [GIVEN]
- [WHEN]
- [THEN]

In the section **[SCENARIO]** you should describe what is the purpose of the test, for example, to check if the field is editable or if the bonus is calculated properly.

The section **[GIVEN]** is responsible for the basic data which needs to be present in the system before executing the test. In the automated tests, you can use the provided libraries to prepare random data. You can have multiple **[GIVEN]** sections - one for one type of data.

In the **[WHEN]** section you describe the action which is tested. For example, opening the page or posting the sales document.

The section **[THEN]** describes the expected result of the test. If the result is negative then the error is shown. To compare the expected behavior and result of [When] you can use the standard library Assert.

HINTS

The sections SCENARIO, GIVEN, WHEN, THEN are not part of the AL language and therefore are added as comments. To add the comments to AL you need to add // in front of it.

TEST LIBRARIES

The tests that you are writing should be data agnostic and should be able to be run in almost an empty database. It means that you should create data for all needed given points in the test. For example customers, items, vendors, etc.

The easiest way to do it is to use libraries that Microsoft is providing. To do that you will need to add the dependency to the **Tests-TestLibraries** extension. It will allow you to create data in just a few lines of code. All libraries are codeunits and contain the procedures that are helpful when writing the tests.

Below there are a few libraries that can be useful for you

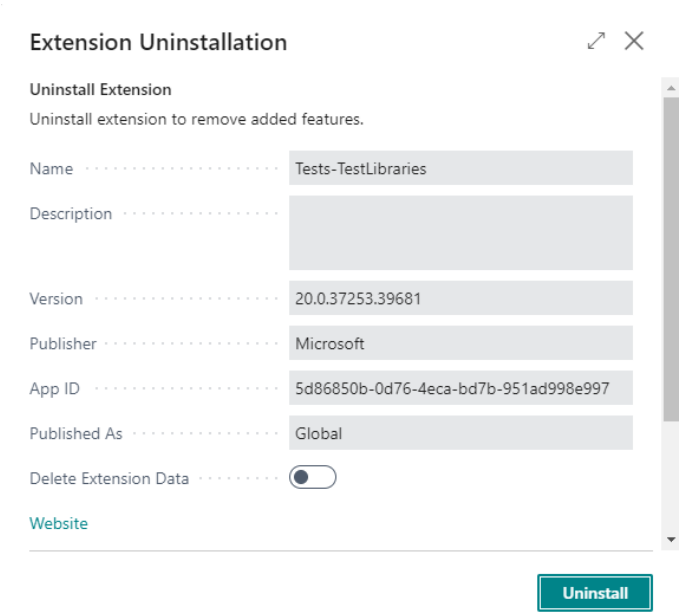
Library – Sales	Contains functions related to **Customers** (such as creating customers, addresses, banks, etc.), **Sales documents** (such as creating a sales order, sales invoice, sales lines, etc.)
Library – Purchase	Contains functions related to **Vendors** (such as creating customers, addresses, banks, etc.), **Purchase documents** (such as creating a purchase order, purchase invoice, purchase lines, etc.)
Library – Inventory	Contains functions related to **Items** (such as creating items, item vendor, etc.)
Library – Warehouse	Contains functions related to Locations (such as creating locations, bins, etc.), Warehouse Documents (such as picking, putting away, etc.)

Library – Random	Contains functions that **allow the creation of random data** such as random integers, code, text, etc.
Library – Utility	Contains functions related to the Number of Series

One of the special kinds of the library is **Library Assert** which allows comparing the results of expected values and current values in the system. This library is a separate extension.

TASK: ADD DEPENDENCIES IN THE TEST APP TO THE TESTS-TESTLIBRARIES

In the test app, you need to add dependencies to the **Tests-TestLibraries** app. For that, you will need to know information which you can find on page **Extension Management**.



1. Open the app.json file for the Test app and find property **dependency**
2. Inside the [] add a new dependency. Add {} and fill all properties. You can find the properties by clicking Ctrl+Space. You need to add the id, name, publisher, and version of the **Tests-TestLibraries**
3. Save the file and go to the Command Pallet (F1) and run function AL: Download Symbols

Solution

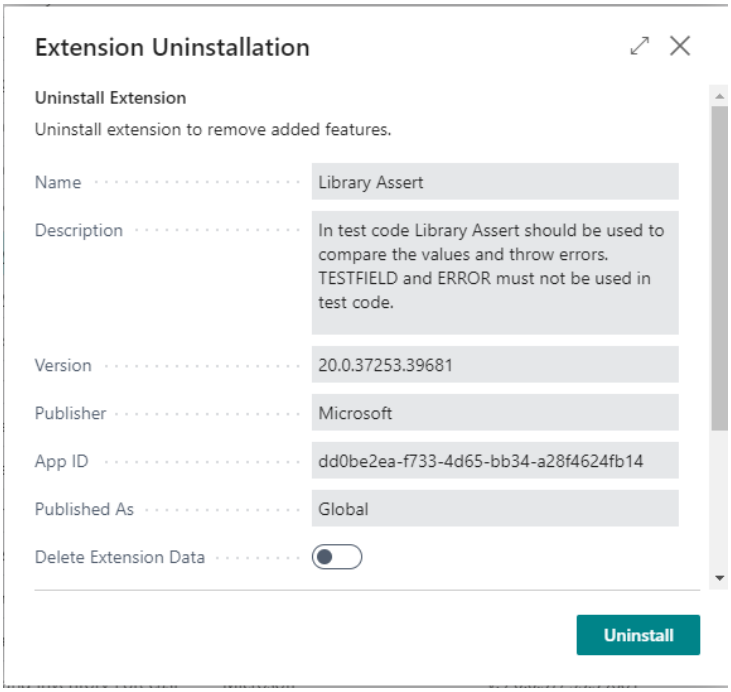
```
"dependencies": [
  {
    "id": "4d35d9fd-22c9-452b-89c5-cf74cd47edda",
    "name": "Bonus Registration",
    "publisher": "myNAVblog.com",
    "version": "1.0.0.0"
  },
  {
    "id": "5d86850b-0d76-4eca-bd7b-951ad998e997",
    "name": "Tests-TestLibraries",
    "publisher": "Microsoft",
    "version": "20.0.37253.39681"
  }
]
```

Hints

In the .alpackage folder, you will see the new file Microsoft_Tests-TestLibraries.app (with the current version).

Task: Add dependencies in the test app to the Library Assert

In the test app, you need to add dependencies to the **Library Assert** app. For that, you will need to know information which you can find on page **Extension Management**.



1. Open the app.json file for the Test app and find property **dependency**
2. Inside the [] add a new dependency. Add {} and fill all properties. You can find the properties by clicking Ctrl+Space. You need to add the id, name, publisher, and version of the **Library Assert**
3. Save the file and go to the Command Pallet (F1) and run function AL: Download Symbols

SOLUTION

```
"dependencies": [
  …
  {
    "id": "dd0be2ea-f733-4d65-bb34-a28f4624fb14",
    "name": "Library Assert",
    "publisher": "Microsoft",
    "version": "20.0.37253.39681"
  }
]
```

HINTS

In the .alpackage folder, you will see the new file Microsoft_Library Assert.app (with the current version).

ASSERTERROR

By default, the tests will fail if the error would be shown. However, in some cases, the error is the expected result of the test. This is why in the line that you expect the error it is needed to add **asserterror**. In such cases, the test would fail if the error would not shown.

TASK: CHECK IF STARTING DATE IS EDITABLE IN THE STATUS RELEASED

It is time to write the first automated test. You will create a simple test that will check if it is possible to change the **Starting Date** after the bonus was released.

1. Create a new file **BonusCardTests.Codeunit.al** and create a new codeunit. You can use snippet **tcodeunit** for that
2. Make sure that the subtype of the codeunit is **Test**
3. Create new global variables in the codeunit and add:
    a. **LibraryRandom**: Codeunit "Library - Random"
    b. **LibraryUtility**: Codeunit "Library – Utility"
    c. **Assert**: Codeunit Assert
4. Create a new procedure and name it (without spaces) "Check If Not Possible To Change Starting Date In Released Status". Remember it must be marked as a **Test**
5. Add the same **[SCENARIO]** (with spaces)
6. Add **[GIVEN]** – "Bouns Header exists in status Released"
7. In the section **[GIVEN]**, init new **MNB Bonus Header**, assign value to the **No.** field. For that use function **GetGlobalNoSeriesCode()** from **Library - Utility** codeunit. Then assign a **Status**

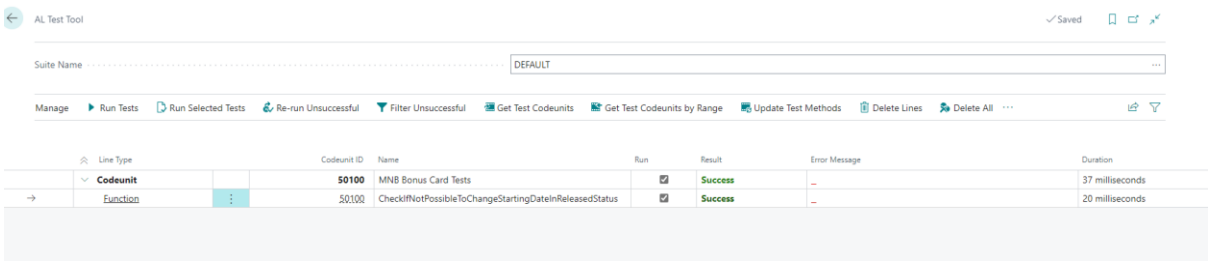
to be **Released** and insert the record (do not make validate and do not run triggers in the Insert method)

8. Add **[WHEN]** - "Validate the Starting Date directly in the code"
9. In the **[WHEN]** section validate the **Starting Date** field with a random value. For that use function **RandDate()** from the **Library - Random** codeunit. Remember that in this place you expect the error so you need to put in the line **asserterror**
10. Add **[THEN]** – "Error is shown that you cannot change the Starting Date in Released status"
11. In the **[THEN]** section add function **ExpectedError()** from the **Assert** codeunit. In the expected error add *'Status cannot be Released'*
12. Publish the extension and open the **AL Test Tool** page
13. Run Action **Get Test Codeunits** and choose your codeunit
14. Run Action **Run Tests** and run all tests

SOLUTION

```
codeunit 50100 "MNB Bonus Card Tests"
{
    Subtype = Test;

    var
        LibraryRandom: Codeunit "Library - Random";
        LibraryUtility: Codeunit "Library - Utility";
        Assert: Codeunit Assert;

    [Test]
    procedure CheckIfNotPossibleToChangeStartingDateInReleasedStatus()
    var
        BonusHeader: Record "MNB Bonus Header";
        StatusCannotBeReleasedErr: Label 'Status cannot be Released';
    begin
        // [SCENARIO] Check If Not Possible To Change Starting Date In
Released Status
        // [GIVEN] Bouns Header exists in status Released
        BonusHeader.Init();
        BonusHeader."No." := LibraryUtility.GetGlobalNoSeriesCode();
        BonusHeader.Status := BonusHeader.Status::Released;
```

```
        BonusHeader.Insert();
        // [WHEN] Validate the Starting Date directly in the code
        asserterror BonusHeader.Validate("Starting Date",
LibraryRandom.RandDate(10));
        // [THEN] Error is shown that you cannot change the Starting
Date in the Released status
        Assert.ExpectedError(StatusCannotBeReleasedErr);
    end;
}
```



HINTS

Try to remove asserterror and see how it looks when an error would occur – your test should fail.

CHECKING FIELD VALUE WITH TESTFIELD()

You can check if the field has proper value and show an error with the if statement and with the error message. This allows you to show the custom error message.

However, there is one method, that can do the same as custom code does. It is called **TestField()**. In the parameters of the method, you define the field which you testing and you can specify an expected value. If the value will be different, then the standard error will be shown.

With this method, you can also test if the field has an empty value. Then in the method parameters, you specify only the field that you are testing.

Examples you can find below.

```
Customer.TestField(Name);

Item.TestField(Blocked, false);
```

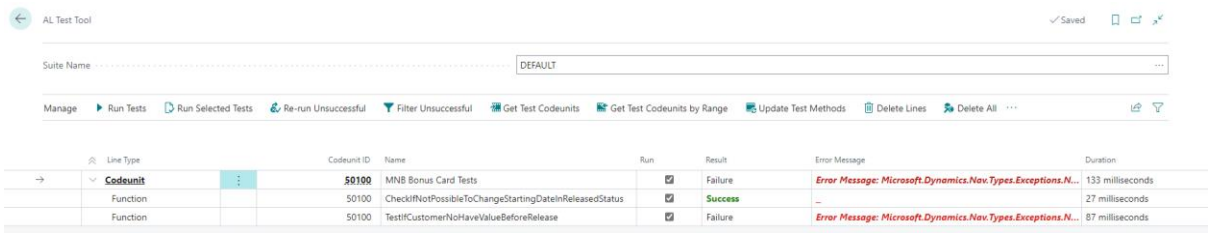
TASK: CHECK IF CUSTOMER NO. IS FILLED VALUE BEFORE RELEASE

It turns out that it is possible to release the bonus without **Customer No.** your system architect said that it should not happen. You need to write the test that checks it and blocks such possibility.

1. Open the file **BonusCardTests.Codeunit.al** and create a new test procedure "Test If Customer No. Have Value Before Release".
2. Add the same **[SCENARIO]** (with spaces)
3. Add **[GIVEN]** – "Bouns Header exists in status Open"
4. In the section **[GIVEN]**, init new **MNB Bonus Header**, assign value to the **No.** field. For that use function **GetGlobalNoSeriesCode()** from **Library - Utility** codeunit. Then assign a Status to be **Open** and insert the record (do not make validate and do not run triggers in the Insert method)
5. Add **[WHEN]** - "Change status to Released"
6. In the **[WHEN]** section validate the **Status** field with a **Released** value. Remember that in this place you expect the error so you need to put in the line **asserterror**
7. Add **[THEN]** – "Error is shown that Customer No. does not have value"
8. In the **[THEN]** section add function **ExpectedError()** from the **Assert** codeunit. In the expected error add '*Customer No. must have a value in Bonus: No.*'
9. Publish the extension and open the **AL Test Tool** page
10. Run Action **Update Test Methods**
11. Run Action **Run Tests** and run all tests

HINTS

Of course, your test at this moment will fail. This is because no code tests if Customer Number is populated when releasing Bonus.



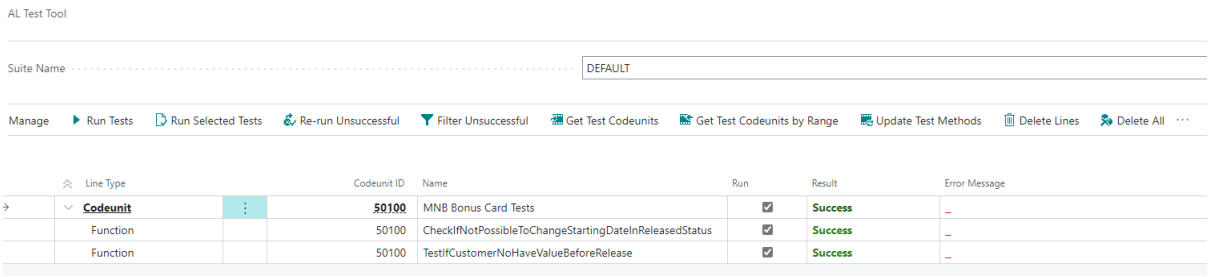
SOLUTION

```
[Test]
procedure TestIfCustomerNoHaveValueBeforeRelease()
var
 BonusHeader: Record "MNB Bonus Header";
 CustomerNoIsEmptyErr: Label 'Customer No. must have a value in Bonus:
No.';
begin
  // [SCENARIO] Test If Customer No. Have Value Before Release
  // [GIVEN] Bouns Header exists in status Open
  BonusHeader.Init();
  BonusHeader."No." := LibraryUtility.GetGlobalNoSeriesCode();
  BonusHeader.Status := BonusHeader.Status::Open;
  BonusHeader.Insert();
  // [WHEN] Change status to Released
  asserterror BonusHeader.Validate(Status,
BonusHeader.Status::Released);
  // [THEN] Error is shown that Customer No. does not have value
  Assert.ExpectedError(CustomerNoIsEmptyErr);
end;
```

12. Open file **BonusHeader.Table.al** and add new triggers **OnValidate()** for **Status** field
13. Create a new local procedure **TestOnRelease()** and add it to the **Status** field trigger
14. In the procedure check if the current status is Released and if not exit the procedure
15. If the status is Released test if the field **Customer No.** does have any value. Use for that **TestField()**
16. Publish the changes and try to run the test one more time

SOLUTION

```
field(5; Status; Enum "MNB Bonus Header Status")
{
   DataClassification = CustomerContent;
   Caption = 'Status';
   trigger OnValidate()
   begin
     TestOnRelease();
   end;
}
…

local procedure TestOnRelease()
    begin
        if Status <> Status::Released then
            exit;
        TestField("Customer No.");
    end;
```



HINTS

You can try yourself and write similar tests for Starting Date, Ending Dat. Also, you can try to write code and test to that check if there is any line on the bonus before the release of the bonus.

CHAPTER SUMMARY

- ✓ In this chapter, you understood how to develop automated tests.
- ✓ You know how to use TestField() method.
- ✓ You added the code to check if the field has value before releasing the bonus.

CHAPTER 9

REPORTS AND REPORTS LAYOUT (WORD, EXCEL)

OBJECTIVES

In this chapter, you will get an overview of how to develop the reports in Word and Excel. The objectives are: Understand why you need to write automated tests

- ✓ Get information on how to create a report object
- ✓ Get familiar with Word layout report
- ✓ Get familiar with Excel layout report
- ✓ Develop a report showing the bonus entries

REPORTS OVERVIEW

A report is an object where you can present data from multiple tables. Business Central reports are used to show the data in a custom way (for example the **Top 10 Customers** report), but also to print documents (for example **Sales Invoice**).

When developing the report you can choose if the layout of the report is based on Word, Excel, or RDLC template. In this chapter, you will get familiar with how to build the report using Microsoft Word and Microsoft Excel. However, the structure of the report is the same for all types of layouts – they are different in presenting the data.

Without additional development, the user can see the preview of the report on the screen, print it, or send it to one of the formats: PDF, XML, Word, or Excel.

There is also a special type of report which does not have got the layout but only processes the data in the background. For example, **Calculate Depreciation**.

A report in AL language contains:

- ✓ Report properties
- ✓ Dataitems with columns
- ✓ Report triggers
- ✓ Global variables
- ✓ Request page

REPORT PROPERTIES

The report properties are added for the whole object. Below you can find the properties which you will need to know when starting development for the AL language.

ApplicationArea	This property is mandatory. It says in which area of the system report will be visible.
Caption	Caption for the report. It should not contain the prefix or suffix.

DefaultLayout	With this property, you can decide if the report has got Word, Excel, or RDLC layout.
RDLCLayout, WordLayout, ExcelLayout	In one of those properties (depends on the **DefaultLayout** property) you need to specify the file path in your extension where the layout file is located.
UsageCategory	If the report should be visible from Tell Me functionality, this property is mandatory. Additionally, you will need to fill the **ApplicationArea** property if you want that report will be seen in the Tell Me.
ProcessingOnly	With this property, you can decide if the report shows any layout or only processes the data.

DATAITEMS AND COLUMNS

In the dataset of the report, you can define the **dataitems**. Each of dataitem represents the set of records from the table. Inside the **dataitem**, you can specify columns or other **dataitem** (nested dataitem for example to show lines of the document).

The columns in the **dataitem** represent the fields from the table. You can also put in the columns the variables which you define as globals.

If you specify the **dateitem** in another dataitem, the report will run it for all records which are defined in the **dataset**. To link to dataitems you need to set the property **DataItemLink** where you specify how the two dataitems are connected. For example, you can use it to show for the **Sales Header** only **Sales Lines** where **Document No.** is the same as **No.** in the header.

In the table below you can find other useful properties for the dataitems.

RequestFilterFields	You can define the fields in which the user can define the filters. The fields are shown on the request page when opening the report.
DataItemTableView	You can define what sorting and filters are set to the dataitem. If you define **DataItemTableView** but you will not define **RequestFilterFields** then the dataitem is not

	shown at all on the requested page. It means that the user cannot add filters for this dataitem.
PrintOnlyIfDetail	If the child dataitem does not have any records in set (is empty) then you can decide with this property if the parent dataitem should be printed or not.

In the **dataitem,** you can define three triggers. The first one is triggered before the dataitem retrieves the data. It is called **OnPreDataItem()** and is triggered only once when the **dataitem** is accessed. You can use it for example to set additional filters with **SetRange()** or **SetFilter()** methods.

The second one, **OnAfterGetRecord()**, is triggered on each record in the set. You can use it to calculate some data. Commonly, most of the code is written in this trigger.

The last one is triggered when exiting from the dataitem. It is called **OnPostDataItem().**

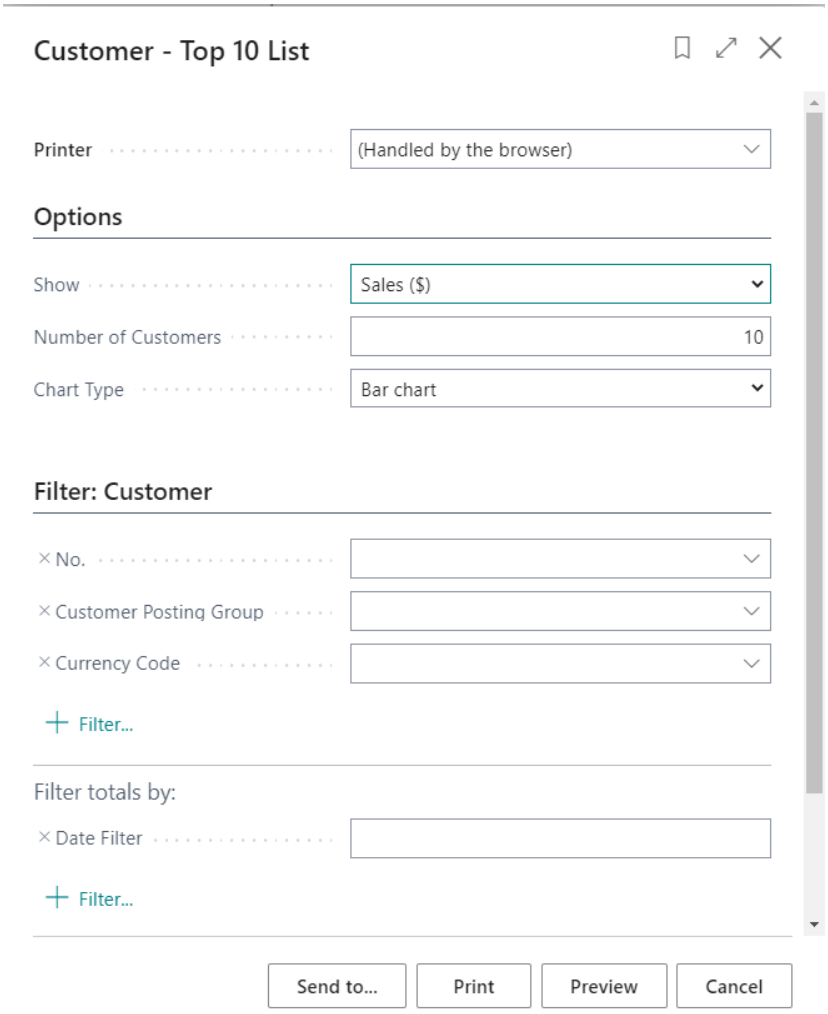
REPORT TRIGGERS

There are three report triggers that you can use when developing the reports. The first one is called **OnInitReport()** and is triggered before the request page is open. The second one is called **OnPreReport()** and is run after closing the request page and before running the dataset. The last one is named **OnPostReport()** and is triggered after the dataset is generated.

REQUEST PAGE

Typically reports have got the request page. This is the special type of page that you define directly in the report. It is presented to the users when the report is opened. You can, similar to the standard page, add the controls and the triggers in the same way. However, in most cases, this page contains only global variables and is not based on any table. You can use this page as the option to run the report.

An example of the request page you can see below.



CalcSums()

Sometimes doing development in the AL language you need to calculate the sum from the records set. You can use the statement **repeat...until** and add the values to one of the variables. However, it is not always the best solution.

Much easier you can do it with the method **CalcSums()**. If you have a decimal or integer field, you can set the filters on the record variable and run the method on a specific field. After that, you can assign to your global variable value of the field. Instead of a value that is in one record, you will get a sum of values in the records set.

The example you can find below.

```
var
    BonusEntry: Record "MNB Bonus Entry";
    AmountSum: Decimal;
begin
    BonusEntry.SetRange("Bonus No.","No.");
    BonusEntry.CalcSums("Bonus Amount");
    AmountSum := BonusEntry."Bonus Amount";
end;
```

TASK: BONUS OVERVIEW REPORT

The system architect would like to get a simple report which shows the information from the header of the Bonus Card such as Number, Customer No., Starting Date, and Ending Date. Also, the report has to show the entries for the bonus and present the total amount of the bonus granted.

1. Create a new file **BonusOverview.Report.al** and create a new report **MNB Bonus Overview**. You can use it for that snippet **treport**
2. Add the properties to the report such as the **Caption**, **UsageCategory**, and **ApplicationArea**.
3. Add the **dataitem** which is based on the **MNB Bonus Header** table. Put that user see the filters for **Customer No.** and **No.** fields

   ### HINTS

   *You can use the **RequestFilterFields** property for your dataitem to add available filters.*

4. Add the columns to the dataitem which are for fields **No.**, **Customer No.**, **Starting Date**, and **Ending Date**
5. Add child **dataitem** which is based on the **MNB Bonus Entry**. It should be linked with the previous dataitem that it shows only records from the exact **MNB Bonus Header**

   ### HINTS

   *You can use the **DataItemLink** property for your dataitem to add available filters.*

6. Add the **Posting Date** field as a filter for the **MNB Bonus Entry** dataitem
7. Add the columns to the **dataitem** which are for fields **Document No., Item No., Posting Date,** and **Bonus Amount**.

HINTS

At this moment the report does not have any layout and will not work. Do not try to run it.

SOLUTION

```
report 65400 "MNB Bonus Overview"
{
    UsageCategory = ReportsAndAnalysis;
    ApplicationArea = All;
    Caption = 'Bonus Overview';

    dataset
    {
        dataitem("MNB Bonus Header"; "MNB Bonus Header")
        {
            RequestFilterFields = "No.", "Customer No.";

            column(No_; "No.")
            {
                IncludeCaption = true;
            }
            column(Customer_No_; "Customer No.")
            {
                IncludeCaption = true;
            }
            column(Starting_Date; "Starting Date")
            {
                IncludeCaption = true;
            }
            column(Ending_Date; "Ending Date")
            {
                IncludeCaption = true;
            }

            dataitem("MNB Bonus Entry"; "MNB Bonus Entry")
            {
                DataItemLink = "Bonus No." = field("No.");
```

```
                RequestFilterFields = "Posting Date";

                column(Document_No_; "Document No.")
                {
                    IncludeCaption = true;
                }
                column(Posting_Date; "Posting Date")
                {
                    IncludeCaption = true;
                }
                column(Item_No_; "Item No.")
                {
                    IncludeCaption = true;
                }
                column(Bonus_Amount; "Bonus Amount")
                {
                    IncludeCaption = true;
                }
            }
        }
    }
}
```

EXCEL LAYOUT

One report can have multiple layouts created. The easiest one to prepare is the Excel Layout. It allows to export data to Microsoft Excel and then use all standard Excel tools such as Pivot Tables or graphs to present data to the users.

AL Language will create an Excel file automatically when the layout path will be added to the properties of the report.

TASK: ADD EXCEL LAYOUT TO THE REPORT

1. Open file **BonusOverview.Report.al** and default layout as Excel in the report properties
2. Add **ExcelLayout** property with a proper path to the file
3. Publish your extension

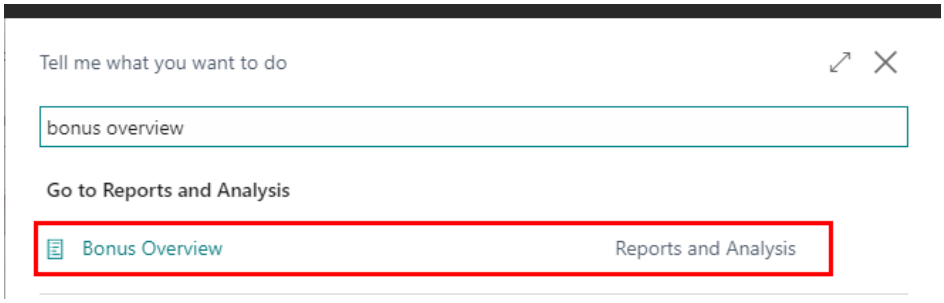
HINTS

You do not need to create a file. The file will be created automatically in the provided path.

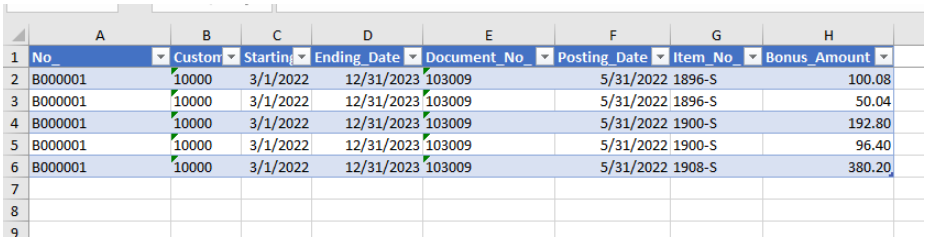
SOLUTION

```
report 65400 "MNB Bonus Overview"
{
    UsageCategory = ReportsAndAnalysis;
    ApplicationArea = All;
    Caption = 'Bonus Overview';
    DefaultLayout = Excel;
    ExcelLayout = './src/Bonus/BonusOverview.xlsx';
```

As a result, a new file should be created. You can run the report inside Business Central.



At this moment the Excel file will have only one Tab – **Data**. The tab cannot be changed, but you can add new tabs and create needed views for analysis.



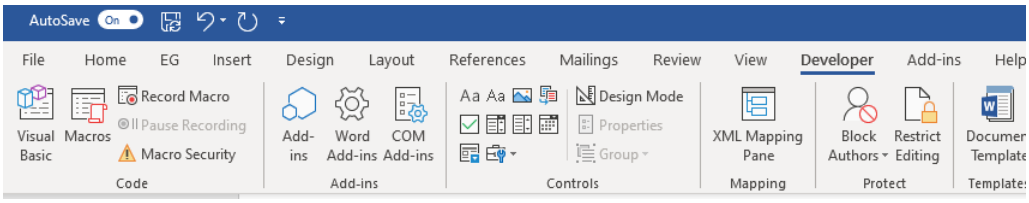
	A	B	C	D	E	F	G	H
1	No_	Custom	Starting	Ending_Date	Document_No_	Posting_Date	Item_No_	Bonus_Amount
2	B000001	10000	3/1/2022	12/31/2023	103009	5/31/2022	1896-S	100.08
3	B000001	10000	3/1/2022	12/31/2023	103009	5/31/2022	1896-S	50.04
4	B000001	10000	3/1/2022	12/31/2023	103009	5/31/2022	1900-S	192.80
5	B000001	10000	3/1/2022	12/31/2023	103009	5/31/2022	1900-S	96.40
6	B000001	10000	3/1/2022	12/31/2023	103009	5/31/2022	1908-S	380.20
7								
8								
9								

WORD LAYOUT

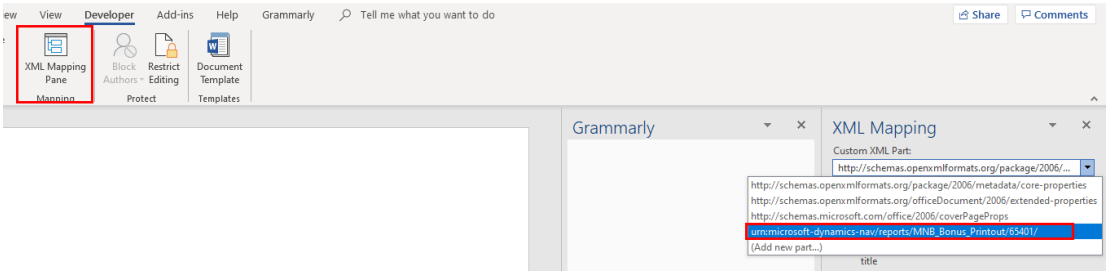
Report Word Layout can be used for reports that should be printed or sent by email. An example can be Sales Invoices, Purchase Orders, or any other document that typically is prepared to be printed.

AL Language will create a Word file automatically when the layout path will be added to the properties of the report.

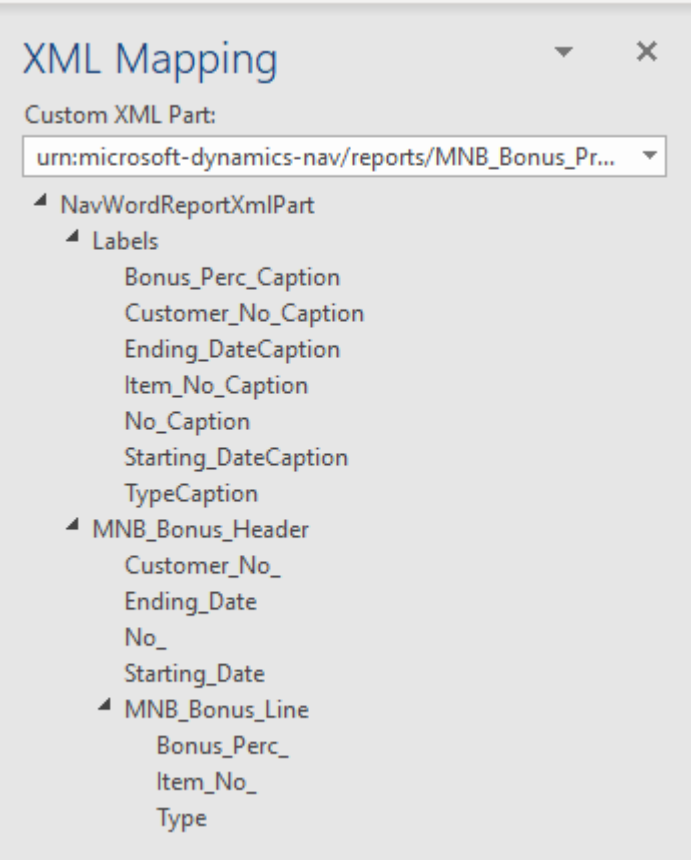
To create Word Layout it is necessary to have the Developer Tab visible in Microsoft Word.



Schema with data from the Business Central report can be found in the XML Mapping Pane.

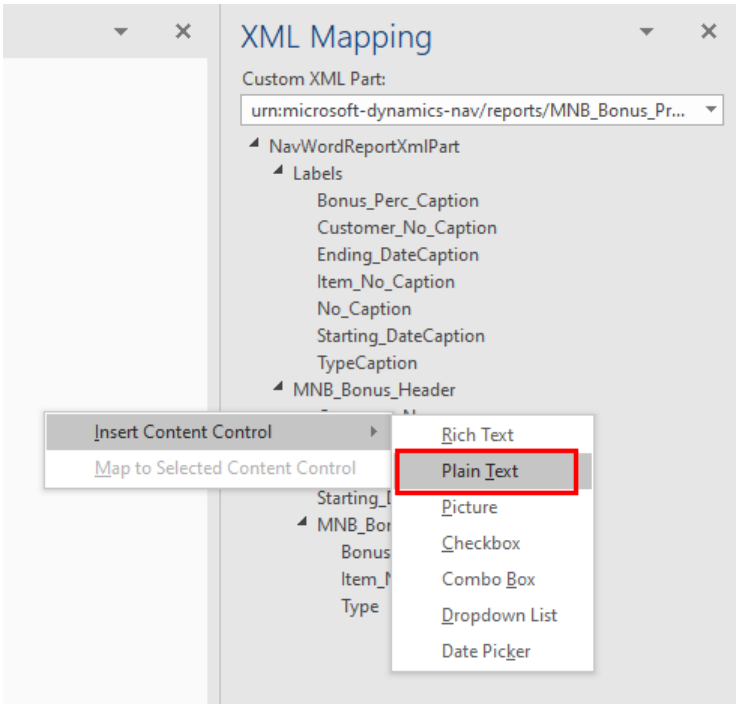


You will be able to find there all columns (fields) that have been added to the report. In the example below you can see the schema for the next task.

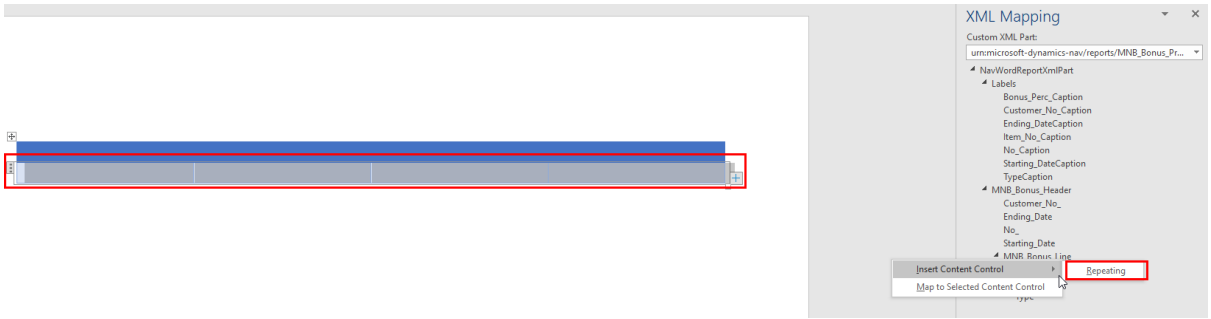


The labels in the report allow you to get the name of the fields without hardcoding the texts. It can be very useful if your report will be used not only in one language. To add the labels to the columns of the report you need to add the property **IncludeCaption**.

To design the report in Word you can use all features that this software gives you. If you need to add the field from the Business Central Dataset you need to add it as **Plain Text**. When printing the control will be replaced with the value from Business Central.



In a Word Layout, it is also possible to create a table that contains multiple lines. To do so, create a simple table in Word, mark the row that should be repeated, and in the **XML Mapping** add the repeater. This is available when you click right mouse click on the dataset item (not on the field)



TASK: BONUS PRINTOUT REPORT

The system architect decided that it should be possible to print in pdf format the report showing the card of the bonus that customer has. The report should be printed from the Bonus Card.

1. Create a new file **BonusPrintout.Report.al** and create a new report **MNB Bonus Printout**. You can use it for that snippet **treport**

2. Add the **Caption** property to the report such as the. This report will be printed from the Bonus Card so **UsageCategory** should be none
3. Add the **dataitem** which is based on the **MNB Bonus Header** table. Put that user see the filters for **Customer No.** and **No.** fields

HINTS

*You can use the **RequestFilterFields** property for your dataitem to add available filters.*

4. Add the columns to the dataitem which are for fields **No.**, **Customer No.**, **Starting Date**, and **Ending Date**
5. Add child **dataitem** which is based on the **MNB Bonus Line**. It should be linked with the previous dataitem that it shows only records from the exact **MNB Bonus Header**

HINTS

*You can use the **DataItemLink** property for your dataitem to add available filters.*

6. Add the columns to the **dataitem** which are for fields **Line Type, Item No., Bonus Perc.**

HINTS

At this moment the report does not have any layout and will not work. Do not try to run it.

SOLUTION

```
report 65401 "MNB Bonus Printout"
{
    UsageCategory = None;
    Caption = 'Bonus Printout';

    dataset
    {
        dataitem("MNB Bonus Header"; "MNB Bonus Header")
        {
            RequestFilterFields = "No.", "Customer No.";

            column(No_; "No.")
```

```
            {
                IncludeCaption = true;
            }
            column(Customer_No_; "Customer No.")
            {
                IncludeCaption = true;
            }
            column(Starting_Date; "Starting Date")
            {
                IncludeCaption = true;
            }
            column(Ending_Date; "Ending Date")
            {
                IncludeCaption = true;
            }

            dataitem("MNB Bonus Line"; "MNB Bonus Line")
            {
                DataItemLink = "Document No." = field("No.");

                column(Type; Type)
                {
                    IncludeCaption = true;
                }
                column(Item_No_; "Item No.")
                {
                    IncludeCaption = true;
                }
                column(Bonus_Perc_; "Bonus Perc.")
                {
                    IncludeCaption = true;
                }
            }
        }
    }
}
```

7. Open the file **BonusPrintout.Report.al** and add a new action in the area Reporting – name it **Print**. Action should run object report **MNB Bouns Printout**

SOLUTION

```
area(Reporting)
{
  action(Print)
  {
    ApplicationArea = All;
    Caption = 'Print';
    Image = Print;
    RunObject = report "MNB Bonus Printout";
    ToolTip = 'Prints bonus card.';
  }
}
```

HINTS

At this moment when clicking the action Bonus No. will not be populated automatically. You will need to choose it manually.

8. Open file **BonusPrintout.Report.al** and default layout as Word in the report properties
9. Add **WordLayout** property with a proper path to the file
10. Publish your extension

HINTS

You do not need to create a file. The file will be created automatically in the provided path.

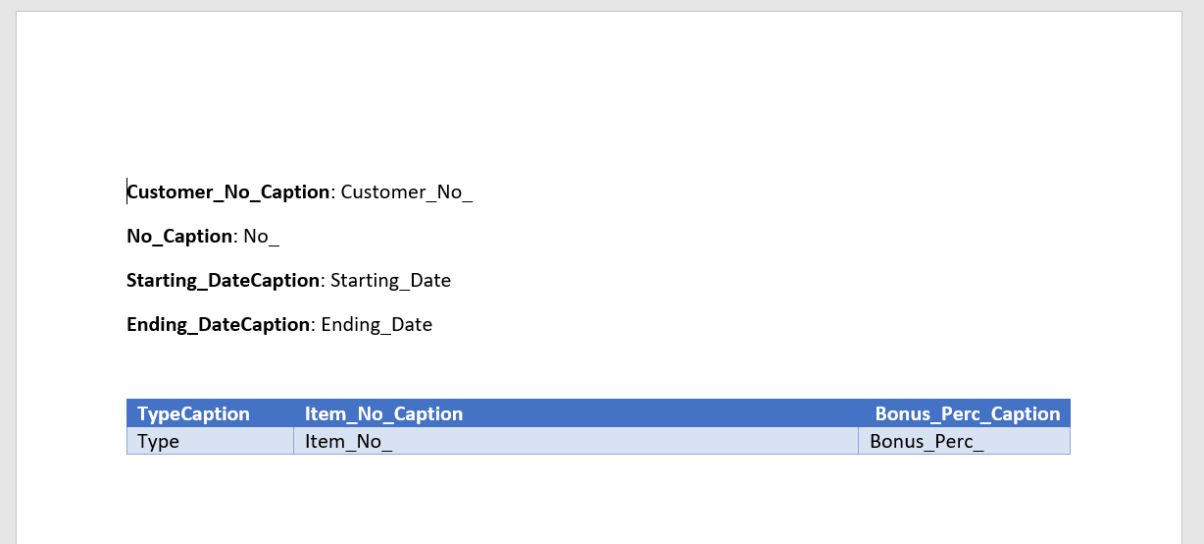
SOLUTION

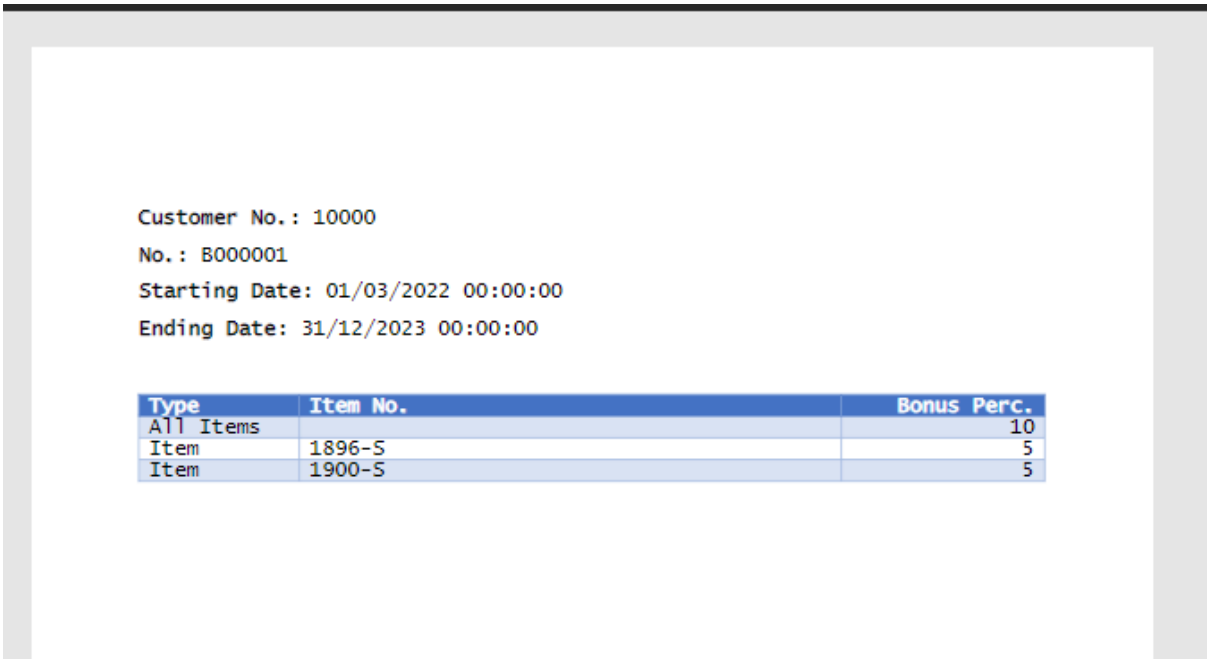
```
report 65401 "MNB Bonus Printout"
{
    UsageCategory = None;
    Caption = 'Bonus Printout';
    DefaultLayout = Word;
    WordLayout = './src/Bonus/BonusPrintout.docx';
```

11. Open created file in Microsoft Word and the fields on the header. Add also lines.

SOLUTION

Remember that the layout can be different in your solution.





CHAPTER SUMMARY

- ✓ In this chapter, you understood how to develop reports for Business Central.
- ✓ You know how to create an Excel and Word Layout
- ✓ You create two reports – one for simple export data and the second for the printout.

CHAPTER 10

ADDITIONAL TASKS FOR THE EXTENSION

OBJECTIVES

In this chapter, you will get additional tasks for the module. The objectives are:

- ✓ Get familiar with how dates work in AL
- ✓ Know when to use a method CalcFields
- ✓ Add FactBox to the list and card page
- ✓ Add Number of Series to the Bonus

WorkDate, Today, Time and CurrentDateTime

If you would like to get the date or time from the system, you can use one of the methods that are present in the AL language. The method **Date** and **Time** give you either the current date or time. The method **CurrentDateTime** gives you both values which you can put to field type **DateTime**.

The method **WorkDate** is widely used across the system. It returns the date which use has set in the settings as a work date.

Task: Information when Bonus has been released

The system architect would like to have a field on the Bonus Card with the information when the last time the bonus has been released.

1. Open a file **BonusHeader.Table.al** and add a new field **Last Released Date**. The field should not be editable and should be type **DateTime** to store the date and time
2. Add the field on the **Bonus Card** page

Solution

```
field(6; "Last Released Date"; DateTime)
{
  DataClassification = SystemMetadata;
  Caption = 'Last Released Date';
  Editable = false;
}
```

```
field("Last Released Date"; Rec."Last Released Date")
{
  ApplicationArea = All;
  ToolTip = 'Specifies the date and time when the bonus was last
released.';
}
```

3. Add the code to the procedure TestOnRelease() that when the Status is Released field **Last Released Date** is filled with the current date and time

SOLUTION

```
local procedure TestOnRelease()
begin
  if Status <> Status::Released then
    exit;
  TestField("Customer No.");
  "Last Released Date" := CurrentDateTime;
end;
```

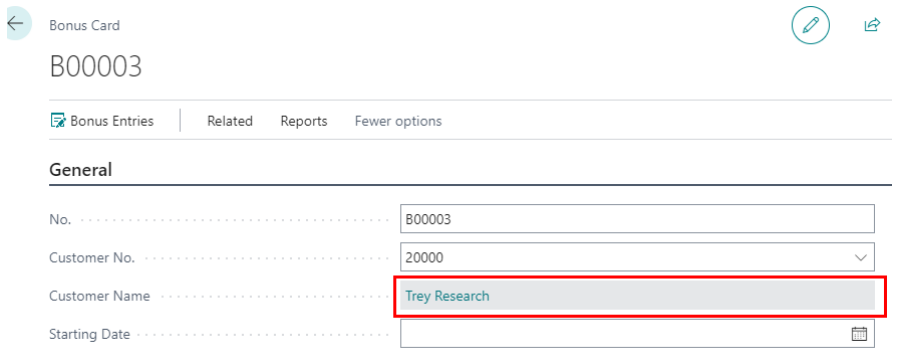
CALCFIELDS

As you know it is possible to add a special type of field - **FlowField**. The field shows the values from different tables for example **Customer Name** based on the chosen **Customer Number**. However, it is not recalculated automatically when the value is typed on the page.

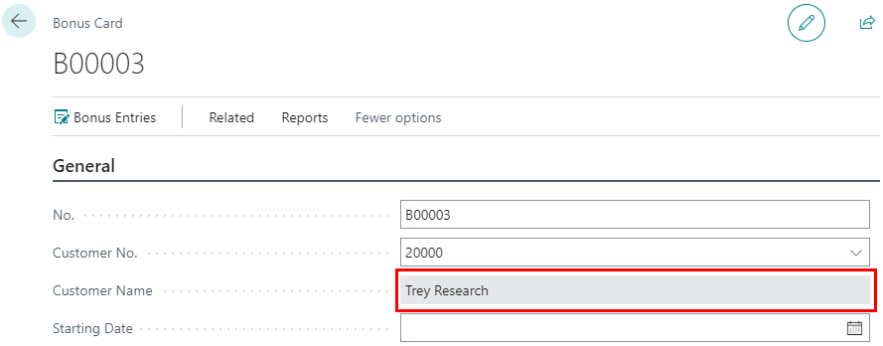
To do so it is needed to use the method **CalcFields()** during the validation of the field.

DRILLDOWN PROPERTY ON THE PAGE FIELD

When you define the FlowField field it is visible on the page as a hover to the data that is presenting for example to show the Customer List with the Customer which is defined in the Customer Name field.



Not always needed that the field should send users to a different record. In such cases on the page field, you can add the property **DrillDown** and set it to **false**.



Task: Add Customer Name to the Bonus Card and List

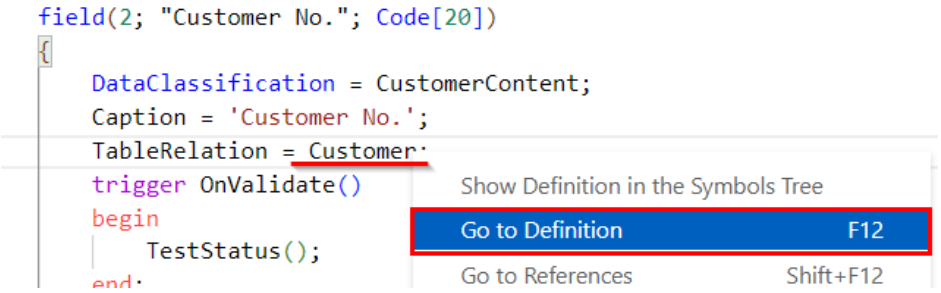
The system architect would like to see on the **Bonus List** and **Bonus Card** the **Customer Name**.

1. Open a file **BonusHeader.Table.al** and add a new field **Customer Name**. The field should not be editable and should be the same type and length as in the Customer table. It should be a FlowField that shows the chosen customer name based on **Customer No.** field

   ### Hints

   To see the table definition you can for example find and variable or table in the table definition and click F12 to see the code in the object. In your solution, you can try it by going to Customer No. field and see Customer table code.

   In this case, the Customer Name is a Text field with a length of 100 characters.



2. Add the field to the **Bonus Card** and the **Bonus List** page. On the **Bonus List,** it should not be possible to drill down to the customer

Solution

```
field(7; "Customer Name"; Text[100])
{
```

```
    Caption = 'Customer Name';
    FieldClass = FlowField;
    CalcFormula  =  lookup(Customer.Name  where("No."  =  field("Customer
No.")));
   Editable = false;
}
```

```
field("Customer Name"; Rec."Customer Name")
{
    ApplicationArea = All;
    ToolTip = 'Specifies the customer name.';
    DrillDown = false;
}
```

HINTS

At this moment your field will work and show proper data but only if you choose the Customer and close the page. In the next step, you will add that it will automatically be recalculated.

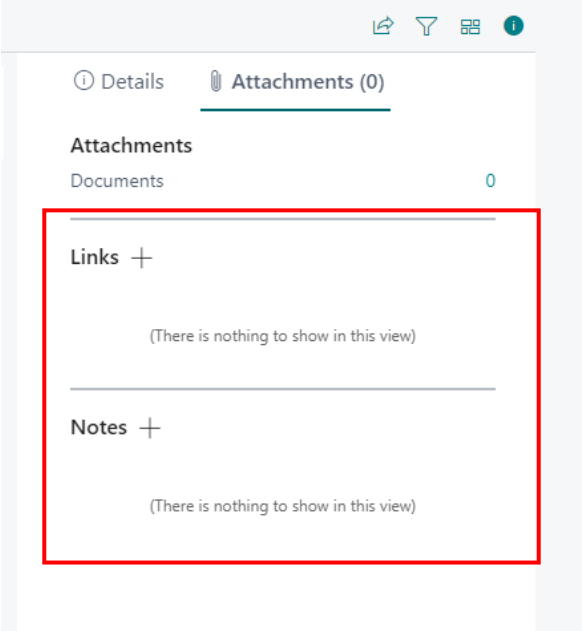
3. Open **BonusHeader.Table.al** file and add the code that onValidate the field **Customer No.** the Customer Name field is recalculated

SOLUTION

```
field(2; "Customer No."; Code[20])
{
    DataClassification = CustomerContent;
    Caption = 'Customer No.';
    TableRelation = Customer;
    trigger OnValidate()
    begin
      TestStatus();
      CalcFields("Customer Name");
    end;
}
```

Notes and Links

In Business Central in many places, users are allowed to add links and notes to the records. You can see them in the FactBox area in the **Attachments**.



Both **Links** and **Notes** can be added to the page as a **systemparts**. There is no need to add any other custom development to handle the logic for notes and links – they work out of the box.

Hints

*Typically the Notes part has **ApplicationArea** set to Notes and Links to RecordLinks. You do not need to add a caption for those parts.*

Task: Add Notes and Links

The system architect decided that the user should be able to add notes and links to the Bonus. You need to add them as FactBoxes to the **Bonus Card**.

1. Open a file **BonusCard.Page.al** and inside the layout section in the bottom add a new area **FactBoxes**
2. Add to it two systemparts – Links and Notes

SOLUTION

```
area(FactBoxes)
{
   systempart(Links; Links)
   {
      ApplicationArea = RecordLinks;
   }
   systempart(Notes; Notes)
   {
       ApplicationArea = Notes;
   }
}
```

HINTS

At this moment both Links and Notes will be shown in the FactBox without group Attachments since there are no other FactBoxes.

TASK: ADD FACTBOX WITH BONUS AMOUNT

At this moment to get the total bonus amount user needs to run the report or export the entries to Excel and calculate the total bonus. The system architect would like to see the total bonus that has been calculated in the FactBox on Bonus Card.

1. Open a file **BonusHeader.Table.al** and add a new field **Bonus Amount**. The field should not be editable. It should be a FlowField that shows the total amount (sum) of Bonus Entries for the chosen Bonus

   ### HINTS

   Remember that type of the field should match the type in the Bonus Entry table.

SOLUTION

```
field(8; "Bonus Amount"; Decimal)
{
   Caption = 'Bonus Amount';
   FieldClass = FlowField;
   CalcFormula = sum("MNB Bonus Entry"."Bonus Amount" where("Bonus No."
= field("No.")));
   Editable = false;
}
```

2. Create a new file **BonusStatistics.Page.al** and set the type to **CardPart**. The page should have the **UsageCategory** set to **None**
3. Make sure that the page is based on the **MNB Bonus Header** table. Add only two fields - **No.** and **Bonus Amount**

SOLUTION

```
page 65404 "MNB Bonus Statistics"
{
    PageType = CardPart;
    UsageCategory = None;
    SourceTable = "MNB Bonus Header";
    Caption = 'Statistics';
    Editable = false;

    layout
    {
        area(Content)
        {
            field("No."; Rec."No.")
            {
                ApplicationArea = All;
                ToolTip = 'Specifies bonus number.';
            }
            field("Bonus Amount"; Rec."Bonus Amount")
            {
                ApplicationArea = All;
                ToolTip = 'Specifies total amout for bonus';
```

```
            }
        }
    }
}
```

4. Open the file **BonusCard.Page.al** and add a new part in the FactBox area. The part should have a unique name (for example Statistics) and it should show the page **MNB Bonus Statistics**. To link the two pages you need to set a property SubPageLink where the Number is the same on both pages

HINTS

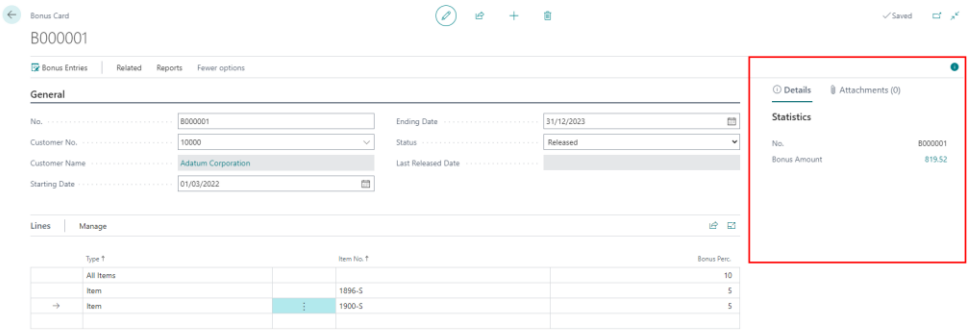
The link between the two pages in this case is based on the No. field because both pages Bonus Card and Bonus Statistics are based on the same table.

SOLUTION

```
area(FactBoxes)
{
  part(Statistics; "MNB Bonus Statistics")
  {
    SubPageLink = "No." = field("No.");
    ApplicationArea = All;
  }
```

HINTS

At this moment both Links and Notes will be shown in the FactBox in the Attachments group and the Statistics will be shown in the Details group.

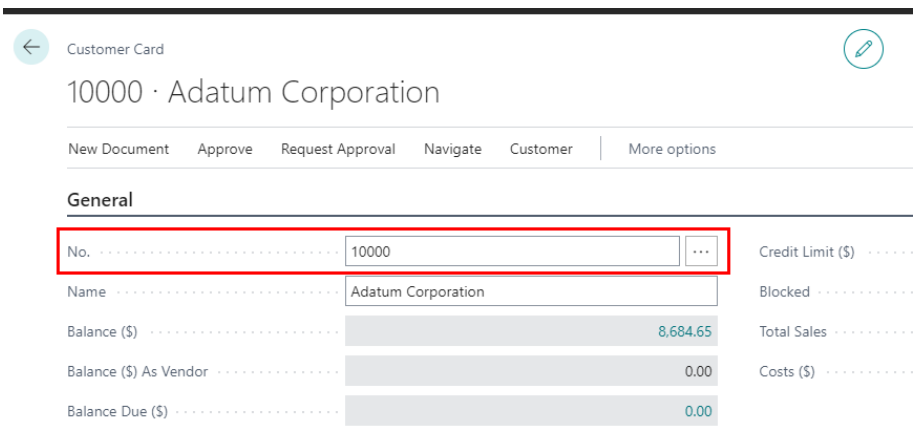


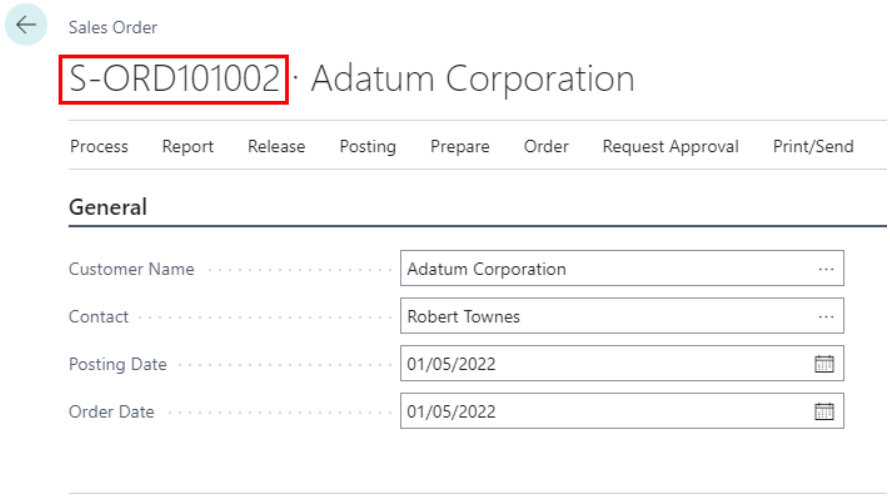
Using Number of Series

In Business Central records such as Customer, Item, Vendor, or documents such as Sales Order, Purchase Invoice, etc., do not have just a simple next number from the integer value in the Number field. Instead, the Number Series are used. The Number Series allows users to use alphanumeric values. It also allows adding in the Number fields special characters.

Typically which Number Series is used for a specific card is defined on the setup table for example Sales & Receivable Setup, Inventory Setup, etc.

Field **No.** across all cards and documents works similarly. For example, the visibility of the field is defined by Number Series used. If the Number Series does not have any relations and it is not allowed to use for the number series manual numbers the field No. would be hidden on the card or document. If the field is visible on the page, it has a defined **AssistEdit** property that allows choosing the related number series.





Checking if the value in the field has changed (Rec and xRec)

If you write a code in the table, to get the current value of the field you just write the name of the field. It is the same as writing before the field name word **Rec**. For example to use field **No.** you can either use "No." **"No."** or **Rec."No."**.

You can also use the value of the field which was before the modification. For that, you need to use **xRec** instead od **Rec**. For example, to get the value of the **"No."** field before modification you can use **xRec."No."**.

Task: Add Number Series to the Bonus

For now, field No. needs to be populated manually. The system architect would like that this field will be populated automatically from the defined Number Series. The definition of the Number Series should be added to the **Sales & Receivable Setup**.

1. In the **src** folder create a new folder **Setup.** Add to new file **SalesReceivablesSetup.TableExt.al**
2. Create a table extension to the **Sales & Receivables Setup** table and add a field **MNB Bonus Nos.** The field should have a table relation to **No. Series** table

Hints

Remember about prefixes when creating the object.

All No. Series have are type Code and have a maximum of 20 characters.

SOLUTION

```
tableextension 65401 "MNB Sales & Receivables Setup" extends "Sales &
Receivables Setup"
{
    fields
    {
        field(65400; "MNB Bonus Nos."; Code[20])
        {
            Caption = 'Bonus Nos.';
            DataClassification = SystemMetadata;
            TableRelation = "No. Series";
        }
    }
}
```

3. In the same folder create a new file **SalesReceivablesSetup.PageExt.al** and create a page extension to **Sales & Receivables Setup** page

4. Add a field **MNB Bonus Nos.** as the last field in the **Number Series**

SOLUTION

```
pageextension 65401 "MNB Sales & Receivables Setup" extends "Sales &
Receivables Setup"
{
    layout
    {
        addlast("Number Series")
        {
            field("MNB Bonus Nos."; Rec."MNB Bonus Nos.")
            {
                ApplicationArea = Basic, Suite;
                ToolTip = 'Specifies the code for the number series that
will be used to assign numbers to bonuses.';
            }
        }
    }
}
```

5. Open the file **BonusHeader.Table.al** and add trigger **OnInsert()**. In the trigger, if the field **No.** is empty then you should:

    a. Retrieve (Get) record from **Sales & Receivables Setup** table

    b. Check if the MNB Bonus Nos. field has any value

    c. Use function from **InitSeries** from **NoSeriesManagement** codeunit

HINTS

When Getting the record from setup tables such as Sales & Receivables Setup you do not need to add any value between (). This is because setup tables have only one record with an empty value.
To check the value if has anything inside you can use TestField.

SOLUTION

```
trigger OnInsert()
var
  SalesSetup: Record "Sales & Receivables Setup";
  NoSeriesManagement: Codeunit NoSeriesManagement;
begin
  if "No." = '' then begin
    SalesSetup.Get();
    SalesSetup.TestField("MNB Bonus Nos.");
    NoSeriesManagement.InitSeries(SalesSetup."MNB Bonus Nos.",
SalesSetup."MNB Bonus Nos.", WorkDate(), "No.", SalesSetup."MNB Bonus
Nos.");
  end;
end;
```

6. Open the file **BonusHeader.Table.al** on validating the field **No.** if the current value of the field (Rec) is different than the previous value of the field (xRec) you should:

    a. Retrieve (Get) record from **Sales & Receivables Setup** table

    b. Use function **TestManual** from **NoSeriesManagement** codeunit to check if manual numbers are allowed in the field

HINTS

Add the code to the local procedure TestNoSeries for more readability.

SOLUTION

```
field(1; "No."; Code[20])
{
   DataClassification = CustomerContent;
   Caption = 'No.';

   trigger OnValidate()
   begin
     TestNoSeries();
   end;
}

…

local procedure TestNoSeries()
var
   SalesSetup: Record "Sales & Receivables Setup";
   NoSeriesManagement: Codeunit NoSeriesManagement;
begin
   if "No." <> xRec."No." then begin
     SalesSetup.Get();
     NoSeriesManagement.TestManual(SalesSetup."MNB Bonus Nos.");
   end;
end;
```

HINTS

*At this moment the **AssistedEdit** for field **No.** has not been added. This is on purpose since it is more advanced.*

How to do so you can find on https://alguidelines.dev/docs/patterns/no-series/

CHAPTER SUMMARY

- ✓ In this chapter, you understood how to add FactBoxes to the page.
- ✓ You made that now the bonus number will be retrieved automatically from the Number Series.
- ✓ You know how to add FlowFields and how to automatically recalculate them.

LAST WORD

If you made it to this page it means that you probably have a working solution for Bonuses.

Congratulations!

I hope you enjoy these exercises. If you have ideas for new tasks or you would like to contribute to this workbook please do not hesitate and send me an email at kbialowas@bc4all.com. I would love to hear your feedback.