
PROGRAMMING IN AL

FOR MORE ADVANCED DEVELOPERS

VERSION 1

TABLE OF CONTENT

INTRODUCTION	3
INTERFACES	5
VARIANT, RECORDREF, FIELDREF	23
LIST AND DICTIONARY	36
HANDLING ERROR IN AL	50
QUERIES	65
CUSTOM APIS AND EXTERNAL BUSINESS EVENTS	76
BACKGROUND TASKS	90
HANDLING CSV, XML, AND JSON FILES FROM THE CODE	95
USEFUL MODULES FROM SYSTEMAPP	109
LAST WORD	125

INTRODUCTION

In 2019 I wrote the first version of a workbook for Basics of AL. This time I am back to you with the version for more advanced developers. Now that you have mastered the basics of the AL language and fundamental development concepts, you are ready to dive into more topics that will elevate your skills as Dynamics 365 Business Central developer.

In this advanced workbook, I will guide you through essential aspects of Business Central development, such as working with interfaces, creating queries and APIs, and using lists and dictionaries. You will also learn how to use system modules to enhance the functionality and performance of your applications.

This workbook is designed to be an interactive and practical guide. Just like the basics workbook, this continuation emphasizes writing code yourself. Avoid the temptation to copy-paste. By typing out the code, you will reinforce your learning and improve your coding fluency. Make notes and highlight areas you find challenging.

The exercises and examples provided are crafted to simulate real-world scenarios, giving you a practical edge in your development projects.

Remember, this workbook is not just a reference guide but a tool to actively work through. Print it out, write in it, and treat it as your development journal. Your engagement will transform theoretical knowledge into practical expertise.

I encourage you to share your experiences and thoughts on social media platforms such as LinkedIn and Twitter. Your feedback on MyNAVBlog.com is also invaluable and helps improve the material for future developers.

This material remains free for you to use and share within your organization. Embrace the journey of becoming an advanced Business Central developer, and let's get started with these exciting new topics!

Krzysztof Bialowas

www.MyNAVBlog.com

July 2024

PROJECT REPOSITORY

The 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/ALForMoreAdvancedDevelopers**](https://github.com/mynavblog/ALForMoreAdvancedDevelopers)

CHAPTER 1

INTERFACES

OBJECTIVES

In this chapter, you will learn what are the interfaces. The main objectives for this part of the workbook are:

- ✓ Understand what are interfaces and how to use them
- ✓ Create the first interface in AL
- ✓ Create a new implementation of the interface in a separate extension

INTRODUCTION

In AL language, interfaces are like agreements that specify what certain objects should be able to do, without dictating exactly how they do it. They help keep code organized and reusable by allowing different parts of a program to work together smoothly. For example, you could have different codeunits that all perform the same task but in their unique way, as long as they meet the requirements set by the interface. This makes it easier to swap out one implementation codeunit for another without causing problems. Think of it as setting guidelines for how things should work together, rather than specifying every little detail.

To make it simplified, using interfaces in AL language can be compared to using a case statement, as both provide a way to define different behaviors based on certain conditions (example option).

Using the interface the first time can look more complicated however it gives benefits compared to a case statement. Some of the key benefits of using it:

- ✓ **Modular Design**

Interfaces promote modular design by allowing the separation of concerns. Code can be written to reduce dependencies on specific implementations and facilitate easier maintenance and updates.

- ✓ **Code Readability**

By coding to interfaces, you can focus on the essential functionalities and behaviors required by an object, leading to cleaner and more readable code. This clarity aids in understanding and maintaining the code over time.

- ✓ **Code Reusability**

Interfaces enable the creation of reusable code components. Different codeunits implement the same interface, providing a common set of functionalities across various parts of the codebase without duplicating the implementation.

- ✓ **Flexibility and Adaptability**

Interfaces support polymorphism, allowing for flexible and adaptable code. This flexibility is particularly valuable when different implementations need to be substituted seamlessly, enhancing the overall agility of the system.

✓ **Testing and Mocking**

Interfaces facilitate easier testing, as mock objects can be created to implement interfaces during testing scenarios. This allows for thorough testing of individual components in isolation, contributing to a more robust testing strategy.

✓ **Future-Proofing**

Interfaces provide a level of abstraction that can shield the rest of the codebase from changes in specific implementations. This helps in future-proofing the system, as modifications or upgrades to individual components can be made without affecting the overall functionality of the system.

EXAMPLE

A simple example of comparing using a case statement and interface instead can be found below. In this example based on the enum the name of the Greatest Of All Time player in each sport is returned.

When analyzing the simple example approaches think about:

- ✓ How would you add a new option (a new sport) to the code?
- ✓ What is the potential risk of making a development mistake in both approaches?
- ✓ Which parts of the code you would need to change?
- ✓ How would you make both examples extensible from the new extension?
- ✓ How much code review and tests need to be done



EXAMPLE - CASE STATEMENT

Step 1 is to create an enum to show the names of sports.

```
enum 50102 "MNB Goat Sports Case"
{
    Extensible = true;
    value(0; "Basketball")
    {
        Caption = 'Basketball';
    }
    value(1; "Football")
    {
        Caption = 'Football';
    }
}
```



```

    }
    value(2; "F1")
    {
        Caption = 'F1';
    }
    value(3; "Ski Jumping")
    {
        Caption = 'Ski Jumping';
    }
}

```

Step 1 is to create a case statement.

```

local procedure ShowGOATCase()
var
    FootballGOATMsg: Label 'The GOAT of Football is Tom Brady if you
are in the US otherwise it is Pele.';
    BasketballGOATMsg: Label 'The GOAT of Basketball is Michael
Jordan.';
    F1GOATMsg: Label 'The GOAT of F1 is Michael Schumacher.';
    SkiJumpingGOATMsg: Label 'The GOAT of Ski Jumping is Adam
Malysz.';
begin
    case GoatSportsCase of
        GoatSportsCase::Football:
            Message(FootballGOATMsg);
        GoatSportsCase::Basketball:
            Message(BasketballGOATMsg);
        GoatSportsCase::F1:
            Message(F1GOATMsg);
        GoatSportsCase::"Ski Jumping":
            Message(SkiJumpingGOATMsg);
    end;
end;

```



EXAMPLE - INTERFACE

Step 1 is to create an interface that contains the procedures which need to be later implemented.

```

interface "MNB IGoatSports"
{
    procedure GetSportGoatName(): Text;
}

```

Step 2 is to create an enum that implements the interface. It means that based on the value correct implementation (in simple words - codeunit) will be used.

```
enum 50100 "MNB Goat Sports" implements "MNB IGoatSports"
{
    Extensible = true;

    value(0; "Basketball")
    {
        Caption = 'Basketball';
        Implementation = "MNB IGoatSports" = "MNB Basketball Goat";
    }
    value(1; "Football")
    {
        Caption = 'Football';
        Implementation = "MNB IGoatSports" = "MNB Football Goat";
    }
    value(2; "F1")
    {
        Caption = 'F1';
        Implementation = "MNB IGoatSports" = "MNB F1 Goat";
    }
    value(3; "Ski Jumping")
    {
        Caption = 'Ski Jumping';
        Implementation = "MNB IGoatSports" = "MNB Ski Jumping Goat";
    }
}
```

Step 3 is to create a codeunit for each enum value that implements the interface procedures.

```
codeunit 50105 "MNB Basketball Goat" implements "MNB IGoatSports"
{
    procedure GetSportGoatName(): Text;
    var
        BasketballGOATMsg: Label 'The GOAT of Basketball is Michael Jordan.';
    begin
        exit(BasketballGOATMsg);
    end;
}

codeunit 50102 "MNB Football Goat" implements "MNB IGoatSports"
{
    procedure GetSportGoatName(): Text;
    var
```

```

        FootballGOATMsg: Label 'The GOAT of Football is Tom Brady if you
are in the US otherwise it is Pele.';
    begin
        exit(FootballGOATMsg);
    end;
}

codeunit 50103 "MNB F1 Goat" implements "MNB IGoatSports"
{
    procedure GetSportGoatName(): Text;
    var
        F1GOATMsg: Label 'The GOAT of F1 is Michael Schumacher.';
    begin
        exit(F1GOATMsg);
    end;
}

codeunit 50104 "MNB Ski Jumping Goat" implements "MNB IGoatSports"
{
    procedure GetSportGoatName(): Text;
    var
        SkiJumpingGOATMsg: Label 'The GOAT of Ski Jumping is Adam
Malysz.';
    begin
        exit(SkiJumpingGOATMsg);
    end;
}

```

Step 4 is to create a function that will show the proper value (in this case name of the sportsman).

```

local procedure ShowGOATInterface()
var
    IGoatSports: Interface "MNB IGoatSports";
begin
    IGoatSports := GoatSports;
    Message(IGoatSports.GetSportGoatName());
end;

```

HINTS

It is possible to add a default implementation for the enum. This means that if no implementation is specified then the default one will be run without showing the error.

One enum can implement multiple interfaces.

There are code actions that help to create codeunits that easily implement the interfaces.

Implementation codeunits need to have all procedures from the interface.



TASK: CREATE RISK ASSESSMENT SCORES FOR ITEMS

The customer asked to add the new fields on the Item Card to calculate the risk score of using the item. The risk can be calculated in 2 different ways based on the setup on the item card:

- ✓ **Item** – as **Risk Impact** multiplied by **Risk Probability** defined on **Item Card**
- ✓ **Item Category** – as **Risk Impact** defined on **Item Category** and **Risk Probability** defined on **Item Card**

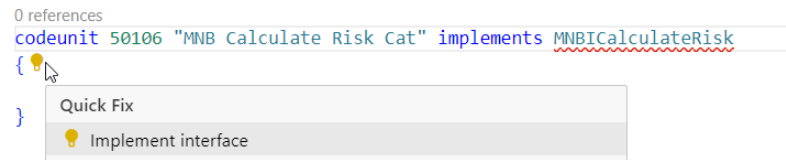
Use the interface to create a functionality to calculate the risk assessment.

1. Create a new file **ItemRiskCalcMethod.Enum.al** in folder **src**. You may also create a subfolder **RiskAssessment**
2. Create a new enum "**MNB Item Risk Calc. Method**" and add two new values **Item** and **Item Category**
3. Create a new file **Item.TabExt.al** and add Table Extension for Item table with 3 new fields:
 - a. **MNB Item Risk Calc. Method** type enum "**MNB Item Risk Calc. Method**"
 - b. **MNB Risk Impact** type integer with minimum value 0
 - c. **MNB Risk Probability** type integer with minimum value 0
 - d. **MNB Risk Score** type integer that is not editable
4. Create Page Extension and add new fields on the bottom of the **Item Card**
5. Create a new file interface **ICalculateRisk.Interface.al** and create a new interface **MNBICalculateRisk** interface using snippet **tinterface**
6. Add new procedure **GetRiskScore** with the parameter for **Item** record



HINT

*Instead of adding procedures manually, it is possible to use code action **Implement Interface***



7. Create a new codeunit "**MNB Calculate Risk Item**" that implements interface **MNBICalculateRisk**
8. Add procedure **GetRiskScore** that calculates Risk Score as Risk Impact multiplied by Risk Probability
9. Open Enum "**MNB Item Risk Calc. Method**" and add that enum implements interface **MNBICalculateRisk**. Add default implementation for the enum "**MNB Calculate Risk Item**". Add the same implementation for value **Item**
10. Add a trigger for all fields in Item Table Extension added previously code that:
 - a. Set Interface based on the value in the field "**MNB Item Risk Calc. Method**"
 - b. Runs function **GetRiskScore**
11. Create a new file **ItemCategory.TabExt.al** and add Table Extension for Item Category table and add new field **MNB Risk Impact** type integer with minimum value 0.
12. Add Page Extension for **Item Category Card** and add a new field
13. Create a new codeunit "**MNB Calculate Risk Category**" that implements interface **MNBICalculateRisk**
14. Add procedure **GetRiskScore** that calculates Risk Score as Risk Impact (from Category) multiplied by Risk Probability (from Item)
15. Open Enum "**MNB Item Risk Calc. Method**" add for value Item Category **MNBICalculateRisk** implantation for interface "**MNB Calculate Risk Category**"
16. Remember to add the function to Calculate the risk on the validation of proper fields



SOLUTION

```
tableextension 50100 "MNB Item" extends Item
{
    fields
    {
        field(50100; "MNB Item Risk Calc. Method"; Enum "MNB Item Risk
Calc. Method")
        {
            DataClassification = CustomerContent;
            Caption = 'Risk Calculation Method';
            trigger OnValidate()
            var
                CalculateItemRiskScore: Codeunit "MNB Calculate Item
Risk Score";
            begin
                CalculateItemRiskScore.SetItemRiskScore(Rec);
            end;
        }
        field(50101; "MNB Risk Impact"; Integer)
        {
            DataClassification = CustomerContent;
            MinValue = 0;
            Caption = 'Risk Impact';
            BlankZero = true;
            trigger OnValidate()
            var
                CalculateItemRiskScore: Codeunit "MNB Calculate Item
Risk Score";
            begin
                CalculateItemRiskScore.SetItemRiskScore(Rec);
            end;
        }
        field(50102; "MNB Risk Probability"; Integer)
        {
            DataClassification = CustomerContent;
            MinValue = 0;
            Caption = 'Risk Probability';
            BlankZero = true;
            trigger OnValidate()
            var
                CalculateItemRiskScore: Codeunit "MNB Calculate Item
Risk Score";
            begin
                CalculateItemRiskScore.SetItemRiskScore(Rec);
            end;
        }
    }
}
```

```

        field(50103; "MNB Risk Score"; Integer)
        {
            DataClassification = CustomerContent;
            MinValue = 0;
            Caption = 'Risk Score';
            BlankZero = true;
            Editable = false;
        }
    }
}

```

```

pageextension 50100 "MNB Item Card" extends "Item Card"
{
    layout
    {
        addlast(content)
        {
            group("MNB Risk Assessment")
            {
                Caption = 'Risk Assessment';
                field("MNB Item Risk Calc. Method"; Rec."MNB Item Risk
Calc. Method")
                {
                    ApplicationArea = All;
                    ToolTip = 'Select the method to calculate the risk
score for the item.';
                }
                field("MNB Risk Impact"; Rec."MNB Risk Impact")
                {
                    ApplicationArea = All;
                    ToolTip = 'Enter the risk impact for the item.';
                }
                field("MNB Risk Probability"; Rec."MNB Risk
Probability")
                {
                    ApplicationArea = All;
                    ToolTip = 'Enter the risk probability for the
item.';
                }
                field("MNB Risk Score"; Rec."MNB Risk Score")
                {
                    ApplicationArea = All;
                    ToolTip = 'Displays the calculated risk score for
the item.';
                }
            }
        }
    }
}

```

```
}  
}
```

```
tableextension 50106 "MNB Item Category" extends "Item Category"  
{  
    fields  
    {  
        field(50100; "MNB Risk Impact"; Integer)  
        {  
            DataClassification = CustomerContent;  
            MinValue = 0;  
            Caption = 'Risk Impact';  
            BlankZero = true;  
        }  
    }  
}
```

```
pageextension 50107 "MNB Item Category Card" extends "Item Category  
Card"  
{  
    layout  
    {  
        addlast(General)  
        {  
            field("MNB Risk Impact"; Rec."MNB Risk Impact")  
            {  
                ApplicationArea = All;  
                ToolTip = 'Enter the risk impact for the item  
category.';  
            }  
        }  
    }  
}
```

```
interface MNBICalculateRisk  
{  
    procedure GetRiskScore(var Item: Record Item);  
}
```

```
codeunit 50101 "MNB Calculate Risk Item" implements MNBICalculateRisk  
{  
    procedure GetRiskScore(var Item: Record Item)
```



```

begin
    Item."MNB Risk Score" := Item."MNB Risk Impact" * Item."MNB Risk
Probability";
end;
}

```

```

codeunit 50106 "MNB Calculate Risk Cat" implements MNBICalculateRisk
{
    procedure GetRiskScore(var Item: Record Item)
    var
        ItemCategory: Record "Item Category";
    begin
        if not ItemCategory.Get(Item."Item Category Code") then begin
            Item."MNB Risk Score" := 0;
            exit;
        end;
        Item."MNB Risk Impact" := 0;

        Item."MNB Risk Score" := ItemCategory."MNB Risk Impact" *
Item."MNB Risk Probability";
    end;
}

```

```

codeunit 50100 "MNB Calculate Item Risk Score"
{
    procedure SetItemRiskScore(var Item: Record Item)
    var
        ICalculateRisk: Interface MNBICalculateRisk;
    begin
        ICalculateRisk := Item."MNB Item Risk Calc. Method";
        ICalculateRisk.GetRiskScore(Item);
    end;
}

```

```

enum 50101 "MNB Item Risk Calc. Method" implements MNBICalculateRisk
{
    Extensible = true;
    DefaultImplementation = MNBICalculateRisk = "MNB Calculate Risk
Item";
    value(0; "Item")
    {
        Caption = 'Item';
        Implementation = MNBICalculateRisk = "MNB Calculate Risk Item";
    }
}

```

```

value(1; "Item Category")
{
    Caption = 'Item Category';
    Implementation = MNBICalculateRisk = "MNB Calculate Risk Cat";
}
}

```



TASK: SET THE RISK IMPACT FIELD AS NOT EDITABLE

The customer is happy with the functionality but would like to be unable to edit the **Risk Impact** field on the Item Card when the **Risk Calculation Method** is **Item Category**.

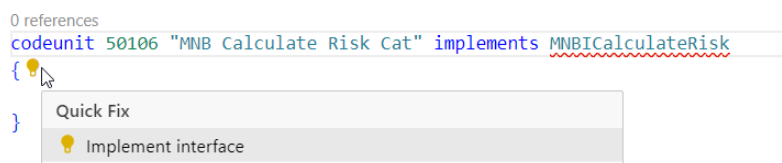
Use the interface to create a functionality to set not editable the risk assessment.

1. Open the **MNBICalculateRisk** interface and add a new function **GetEditableRiskImpact** that returns a boolean
2. Open each implementation codeunit of the interface and add the missing procedure. Set *true* for **Item** implementation and *false* for Item **Category** Implementation



HINT

Instead of adding procedures manually, it is possible to use code action Implement Interface. Remember to turn on showing actions in settings first.



3. In the **Item Card** page extension add a new variable **MNBRiskImpactEditable** and add it as value to the Editable property of the field **MNB Risk Impact**
4. Add a trigger to the field **MNB Item Risk Calc. Method** onValidate to update page
5. Add trigger OnAfterGetCurrRecord to Item Card page extension and set **MNBRiskImpactEditable** variable based on interface **MNBICalculateRisk** procedure **GetEditableRiskImpact**



SOLUTION

```

interface MNBICalculateRisk
{
    procedure GetRiskScore(var Item: Record Item);
    procedure GetEditableRiskImpact(): Boolean;
}

```

```

pageextension 50100 "MNB Item Card" extends "Item Card"

...

field("MNB Item Risk Calc. Method"; Rec."MNB Item Risk Calc. Method")
{
    ApplicationArea = All;
    ToolTip = 'Select the method to calculate the risk
score for the item.';
    trigger OnValidate()
    begin
        CurrPage.Update(true);
    end;
}

field("MNB Risk Impact"; Rec."MNB Risk Impact")
{
    ApplicationArea = All;
    ToolTip = 'Enter the risk impact for the item.';
    Editable = MNBRiskImpactEditable;
}

...

var
    MNBRiskImpactEditable: Boolean;

    trigger OnAfterGetCurrRecord()
    var
        MNBICalculateRisk: Interface MNBICalculateRisk;
    begin
        MNBICalculateRisk := Rec."MNB Item Risk Calc. Method";
        MNBRiskImpactEditable :=
MNBICalculateRisk.GetEditableRiskImpact();
    end;

```



TASK: NEW OPTION TO CALCULATE RISK

The customer asked for a new method of calculating the risk – based on Item Variants. So Risk Score is calculated as **Risk Probability** from the **Item Card** and the average value of Risk Impact from the Item Variant. However, because of the decision in your organization, it was said to put that part in a separate extension.

Use the interface to create a functionality to set not editable the risk assessment.

1. Create a new extension and add dependency to the app created in previous tasks
2. Create a new file **ItemVariant.TabExt.al** and add Table Extension for Item Variant table. Add new field **MNB Risk Impact** type integer with minimum value 0
3. Add Page Extension for **Item Variants** and add a new field
4. Create a new file **Item.TabExt.al** and add Table Extension for Item table. Add a new field **MNB Variants Risk Impact** type integer and field class flowfield that calculates the average from item variants.
5. Add implementation codeunit "**MNB Calculate Risk Variant**" that implements **MNBICalculateRisk** interface. Calculate the risk score for the Item and make not editable field **Risk Impact**.
6. Add enum extension to enum "**MNB Item Risk Calc. Method**" with value **Item Variant** and proper **MNBICalculateRisk** interface implementation



SOLUTION

```
tableextension 60100 "MNB2 Item Variant" extends "Item Variant"
{
    fields
    {
        field(50100; "MNB Risk Impact"; Integer)
        {
            DataClassification = CustomerContent;
        }
    }
}
```

```

        Caption = 'Risk Impact';
        MinValue = 0;
        BlankZero = true;
    }
}
}

```

```

pageextension 60100 "MNB2 Item Variants" extends "Item Variants"
{
    layout
    {
        addlast(Control1)
        {
            field("MNB Risk Impact"; Rec."MNB Risk Impact")
            {
                ApplicationArea = All;
                Tooltip = 'Specifies the risk impact of the item. The
risk impact is used to calculate the risk score of the item.';
            }
        }
    }
}

```

```

tableextension 60101 "MNB2 Item" extends Item
{
    fields
    {
        field(60101; "MNB Variant Risk Impact"; Integer)
        {
            Caption = 'Average Variant Risk Impact';
            FieldClass = FlowField;
            CalcFormula = Average("Item Variant"."MNB Risk Impact"
where("Item No." = field("No.")));
            BlankZero = true;
        }
    }
}

```

```

pageextension 60101 "MNB2 Item Card" extends "Item Card"
{
    layout
    {
        addlast("MNB Risk Assessment")
    }
}

```

```

        field("MNB Variant Risk Impact"; Rec."MNB Variant Risk
Impact")
    {
        ApplicationArea = All;
        Tooltip = 'Specifies average risk impact for item
variant.';
    }
}
}
}

```

```

codeunit 60102 "MNB Calculate Risk Variant" implements MNBICalculateRisk
{
    procedure GetRiskScore(var Item: Record Item)
    begin
        Item.CalcFields("MNB Variant Risk Impact");
        Item."MNB Risk Impact" := 0;
        Item."MNB Risk Score" := Item."MNB Variant Risk Impact" *
Item."MNB Risk Probability";
    end;

    procedure GetEditableRiskImpact(): Boolean
    begin
        exit(false);
    end;
}

```

```

enumextension 60100 "MNB2 Item Risk Calc. Method" extends "MNB Item Risk
Calc. Method"
{
    value(60100; "MNB 2Item Variant")
    {
        Caption = 'Item Variant';
        Implementation = MNBICalculateRisk = "MNB Calculate Risk
Variant";
    }
}

```

CHAPTER SUMMARY

- ✓ In this chapter, we created new functionality for Risk Assessment on Item Card
- ✓ You developed the interface and codeunits that implement it
- ✓ You also created a new extension that easily allows to implementation interface without changes to the structure of the Risk Assessment functionality but adding new logic

CHAPTER 2

VARIANT, RECORDREF, FIELDREF

OBJECTIVES

When working with Records sometimes it is required to build more generic code which can be achieved using variants, RecordRef, and FieldRef object types. The main objectives for this chapter are:

- ✓ Learn how to use Variants
- ✓ How to use RecordRefs together with FieldRefs
- ✓ How to build a functionality that can be reusable for many tables

VARIANT DATA TYPE

In AL language, the definition of Variant is very simple: variant represents an AL object. It means to variant it is possible to assign any declared variable for example text, code, decimal but also record or even if the variable is codeunit.

Therefore the variant type can be very powerful and allows to building of reusable code. Usage of the Variant in Business Central can be found in **Type Helper** codeunit in procedure Evaluate.

```
procedure Evaluate(var Variable: Variant; String: Text; Format: Text; CultureName: Text): Boolean
begin
    // Variable is return type containing the string value
    // String is input to evaluate
    // Format is in format "MM/dd/yyyy" only supported on date, search MSDN for more details ("CultureInfo.Name Property")
    // CultureName is in format "en-US", search MSDN for more details ("Custom Date and Time Format Strings")
    case true of
        Variable.IsDate:
            exit(TryEvaluateDate(String, Format, CultureName, Variable));
        Variable.IsDateTime:
            exit(TryEvaluateDateTime(String, Format, CultureName, Variable));
        Variable.IsDecimal:
            exit(TryEvaluateDecimal(String, CultureName, Variable));
        Variable.IsInteger:
            exit(TryEvaluateInteger(String, CultureName, Variable));
        else
            Error(UnsupportedTypeErr);
    end;
end;
```

The function allows to variable (either date, datetime, decimal, or integer) based on the string. As you can see you do not need to add any of that types to the parameters – it will return the variant in the **Variable** parameter if it is possible to do so.

RECORDREF DATA TYPE

The RecordRef data type allows to refer to the record from any table. The RecordRef allows you to do standard operations that you can do with records such as read, modify, or even delete. There are some methods related to the RecordRef type that are worth knowing.

Method	Description
Open(Table Number)	Defines to which table the RecordRef will be referring. For Example, Open(Database::Customer) causes the RecordRef to refer to the Customer Table. Note: it will not show data of the table until FindFirst, Get, FindSet or any other method is run after
GetTable(Record)	Gets the table of a record variable and causes the RecordRef to refer to the same table.

	<p>All filters that apply to the Record variable will also be applied to the RecordRef.</p> <p>This method does not require first to run the Open method</p>
Field(Field Number)	<p>Gets information about the Field in the table for which RecordRef is defined.</p> <p>It allows us to get the value of the field but also such parameters as caption or type of the field</p>

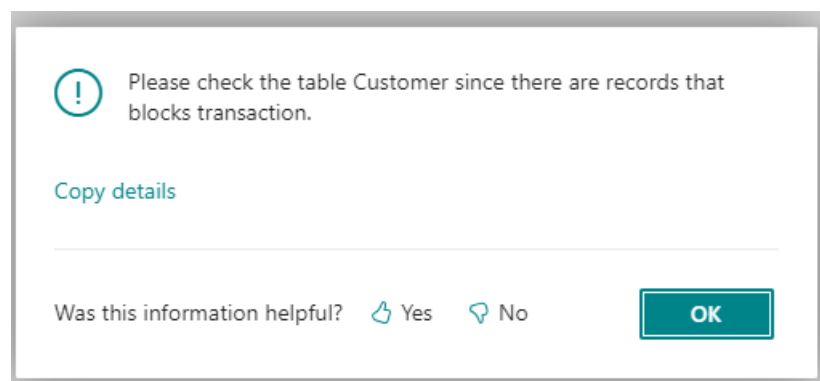


EXAMPLE RECORDREF – OPEN

The below example allows us to show the table caption based on the parameter assigned to the RecordRef.

```
local procedure RecordRefOpenExample()
var
    RecRef: RecordRef;
    TableNo: Integer;
    IsCustomer: Boolean;
    PleaseCheckTableErr: Label 'Please check the table %1 since there are
records that block transactions.';
begin
    IsCustomer := true;
    if IsCustomer then
        TableNo := Database::Customer
    else
        TableNo := Database::Vendor;

    RecRef.Open(TableNo);
    Error(PleaseCheckTableErr, RecRef.Caption);
end;
```





EXAMPLE RECORDREF – GETTABLE

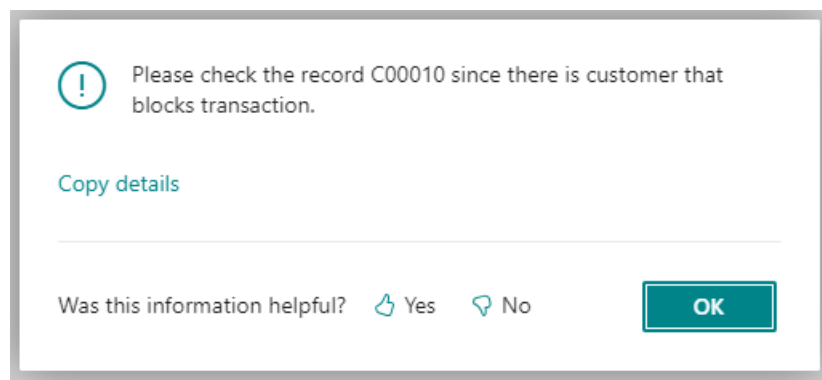
The below example finds the first Record in the Customer Table and assigns that record to the RecordRef variable. At the end, it shows the value of field number 1 in the table.

```
local procedure RecordRefGetExample()  
var  
    RecRef: RecordRef;  
    Customer: Record Customer;  
    IsCustomer: Boolean;  
    PleaseCheckTableErr: Label 'Please check the record %1 since there  
is a customer that blocks the transaction.';  
begin  
    Customer.FindFirst();  
  
    RecRef.GetTable(Customer);  
    Error(PleaseCheckTableErr, RecRef.Field(1).Value);  
end;
```



NOTE

This is just an example and you never should hardcode in the code number of the field or values.



FIELDREF DATA TYPE

The FieldRef, similar to the RecordRef, allows to refer to any field in the given table. Both types work together to get data or do any operations. It also allows to modify values in the field.



EXAMPLE FIELDREF – VALUE

Below example assign the first record from the Customer table to RecordRef and then assign the Description field (number 2) to the FieldRef variable. In the end, it changes the value of the Description field and modifies the record.

```
local procedure FieldRefModifyNameExample()
var
    RecRef: RecordRef;
    FldRef: FieldRef;
    Customer: Record Customer;
    IsCustomer: Boolean;
    PleaseCheckTableErr: Label 'Please check the record %1 since there
is a customer that blocks the transaction.';
begin
    Customer.FindFirst();

    RecRef.GetTable(Customer);
    FldRef := RecRef.Field(2);
    FldRef.Value('New Name of Customer');
    RecRef.Modify();
end;
```



NOTE

This is just an example and you never should hardcode in the code number of the field or values.



TASK: CHECKING IF MANDATORY FIELDS HAVE VALUE

The customer asked for the functionality that checks if all mandatory fields are fields on the Vendor and the Customer Card. The functionality should have one flexible setup. The user should be able to click the action Check Mandatory Fields on the card to see if there are no errors. Note that functionality should work only for the Code and Text fields at this moment.

1. Create a new file **MandatoryFieldSetup.Tab.al** and add table "**MNB Mandatory Field Setup**". Add below fields:
 - a. **Table No.** – Integer, not blank with minimum value 0

- b. **Table Name** – Text, Calcfield showing the table name from AllObjWithCaption table related to the proper table
 - c. **Field No.** – Integer, not blank with minimum value 0
 - d. **Field Caption** – Text Calcfield showing the field caption from the Field table related to the proper field in the table
2. Add a Page for **Mandatory Field Setup** and add all fields.

- **HINT**

*You may add the OnAssistEdit triggers to show the list of the tables and list of the fields. You can use it to list all tables on the **Objects** page and record **AllObjWithCaption**. To select a field number you can use codeunit **Field Selection** and record **Field**. An example of the code you can find in the solution below or the repository.*

3. Create a codeunit "**MNB Mandatory Field Check**" and create an internal function **TestMandatoryFields**. As the parameter for the function add Variant which will represent the record to check.
4. In the function check first if the variant is recorded and then GetTable using RecordRef. Add code that checks if there are mandatory fields for the Table in a setup table and then check if there are any fields with empty values that are set as mandatory. If any field exists without value show an error with a caption of the field.
5. Add a new page extension for **Customer Card** and add action **MNBCheckMandatoryFields**. Action should run the function from the created codeunit.
6. Add a new page extension for **Vendor Card** and add action **MNBCheckMandatoryFields**. Action should run the function from the created codeunit.



SOLUTION

```
table 50108 "MNB Mandatory Field Setup"
{
    Caption = 'Mandatory Field Setup';
    DataClassification = CustomerContent;

    fields
    {
        field(1; "Table No."; Integer)
        {
            Caption = 'Table No.';
            NotBlank = true;
            BlankZero = true;
            MinValue = 0;
            trigger OnValidate()
            begin
                CalcFields("Table Name");
            end;
        }
        field(2; "Table Name"; Text[249])
        {
            Caption = 'Table Name';
            FieldClass = FlowField;
            CalcFormula = lookup(AllObjWithCaption."Object Caption"
where("Object Type" = const(Table), "Object ID" = field("Table No.")));
        }
        field(3; "Field No."; Integer)
        {
            Caption = 'Field No.';
            NotBlank = true;
            BlankZero = true;
            trigger OnValidate()
            var
                TableNoErr: Label 'Table No. must have a value.';
                FieldNoErr: Label 'Field No. must have a value.';
            begin
                if "Table No." = 0 then
                    Error(TableNoErr);
                if "Field No." = 0 then
                    Error(FieldNoErr);
                CheckIfFieldIsTextField();
                CalcFields("Field Caption");
            end;
        }
        field(4; "Field Caption"; Text[100])
```

```

    {
        Caption = 'Field Caption';
        FieldClass = FlowField;
        CalcFormula = lookup(Field."Field Caption" where(TableNo =
field("Table No."), "No." = field("Field No.")));
    }
}

keys
{
    key(PK; "Table No.", "Field No.")
    {
        Clustered = true;
    }
}

local procedure CheckIfFieldIsTextField()
var
    SelectedField: Record Field;
    TypeErr: Label 'Field must be a Text or Code field.';
begin
    SelectedField.Get("Table No.", "Field No.");
    if not (SelectedField.Type in [SelectedField.Type::Text,
SelectedField.Type::Code]) then
        Error(TypeErr);
end;
}

```

```

page 50109 "MNB Mandatory Field Setup"
{
    PageType = List;
    SourceTable = "MNB Mandatory Field Setup";
    ApplicationArea = All;
    UsageCategory = Administration;
    Caption = 'Mandatory Field Setup';

    layout
    {
        area(Content)
        {
            repeater(General)
            {
                field("Table No."; Rec."Table No.")
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies the table number of the table
that contains the field that you want to make mandatory.';

```

```

        trigger OnAssistEdit()
        var
            AllObjWithCaption: Record AllObjWithCaption;
            ObjectAndFieldsMgt: Codeunit "MNB Object And
Fields Mgt.";
        begin
            if
ObjectAndFieldsMgt.SelectObject(AllObjWithCaption) then
                Rec.Validate("Table No.",
AllObjWithCaption."Object ID");
            end;
        }
        field("Table Name"; Rec."Table Name")
        {
            ApplicationArea = All;
            ToolTip = 'Specifies the name of the table that
contains the field that you want to make mandatory.';
        }
        field("Field No."; Rec."Field No.")
        {
            ApplicationArea = All;
            ToolTip = 'Specifies the field number of the field
that you want to make mandatory.';
            trigger OnAssistEdit()
            var
                SelectedField: Record Field;
                ObjectAndFieldsMgt: Codeunit "MNB Object And
Fields Mgt.";
            begin
                if ObjectAndFieldsMgt.SelectFieldNo(Rec."Table
No.", SelectedField) then
                    Rec.Validate("Field No.",
SelectedField."No.");
                end;
            }
        }
        field("Field Caption"; Rec."Field Caption")
        {
            ApplicationArea = All;
            ToolTip = 'Specifies the caption of the field that
you want to make mandatory.';
        }
    }
}
}
}
}

```



```

codeunit 50107 "MNB Object And Fields Mgt."
{
    /// <summary>
    /// Selects the table from the list of all tables
    /// </summary>
    /// <param name="Result">Object that has been selected</param>
    /// <returns>True if object has been selected</returns>
    internal procedure SelectObject(var Result: Record
AllObjWithCaption): Boolean
    var
        AllObjects: Record AllObjWithCaption;
        Objects: Page Objects;
    begin
        AllObjects.FilterGroup(2);
        AllObjects.SetRange("Object Type", AllObjects."Object
Type":Table);
        AllObjects.FilterGroup(0);

        Objects.SetRecord(AllObjects);
        Objects.SetTableView(AllObjects);
        Objects.LookupMode := true;

        if Objects.RunModal() = Action::LookupOK then begin
            Objects.GetRecord(Result);
            exit(true);
        end;

        exit(false);
    end;

    /// <summary>
    /// Allows to select fields from the table
    /// </summary>
    /// <param name="TableNo">Number of table from which the field
should be selected</param>
    /// <param name="SelectedField">Field that has been selected</param>
    /// <returns>True if field has been selected</returns>

    procedure SelectFieldNo(TableNo: Integer; var SelectedField: Record
Field): Boolean
    var
        FieldSelection: Codeunit "Field Selection";
    begin
        SelectedField.FilterGroup(2);
        SelectedField.SetRange(TableNo, TableNo);
    end;
}

```

```

        SelectedField.SetFilter(Type, '%1|%2', SelectedField.Type::Text,
SelectedField.Type::Code);
        SelectedField.FilterGroup(0);
        if FieldSelection.Open(SelectedField) then
            exit(true);
        exit(false);
    end;
}

```

```

codeunit 50108 "MNB Mandatory Field Check"
{
    internal procedure TestMandatoryFields(RecToCheck: Variant)
    var
        MandatoryFieldsSetup: Record "MNB Mandatory Field Setup";
        RecRef: RecordRef;
        FldRef: FieldRef;
        FieldIsMandatoryErr: Label 'Field %1 is mandatory.', Comment =
'%1 - field caption';
        FieldsAreSetMsg: Label 'All mandatory fields for %1 are filled
in.', Comment = '%1 - table caption';
    begin
        if not RecToCheck.IsRecord then
            exit;

        RecRef.GetTable(RecToCheck);
        MandatoryFieldsSetup.SetRange("Table No.", RecRef.Number);
        if MandatoryFieldsSetup.FindSet() then begin
            repeat
                FldRef:= RecRef.Field(MandatoryFieldsSetup."Field No.");
                if Format(FldRef.Value) = '' then
                    Error(FieldIsMandatoryErr, FldRef.Caption);
            until MandatoryFieldsSetup.Next() = 0;
            Message(FieldsAreSetMsg, RecRef.Caption);
        end;
    end;
}

```

```

pageextension 50101 "MNB Customer Card" extends "Customer Card"
{
    actions
    {
        addlast("F&unctions")
    }
}

```

```

    {
        action(MNBCheckMandatoryFields)
        {
            ApplicationArea = All;
            Image = CheckList;
            Caption = 'Check Mandatory Fields';
            ToolTip = 'Check if all mandatory fields are filled in
on the card.';
            trigger OnAction()
            var
                MandatoryFieldCheck: Codeunit "MNB Mandatory Field
Check";
            begin
                MandatoryFieldCheck.TestMandatoryFields(Rec);
            end;
        }
    }
    addlast(Category_Process)
    {
        actionref(MNBCheckMandatoryFields_Promoted;
MNBCheckMandatoryFields)
        {
        }
    }
}

```

```

pageextension 50102 "MNB Vendor Card" extends "Vendor Card"
{
    actions
    {
        addlast("F&unctions")
        {
            action(MNBCheckMandatoryFields)
            {
                ApplicationArea = All;
                Image = CheckList;
                Caption = 'Check Mandatory Fields';
                ToolTip = 'Check if all mandatory fields are filled in
on the card.';
                trigger OnAction()
                var
                    MandatoryFieldCheck: Codeunit "MNB Mandatory Field
Check";
                begin
                    MandatoryFieldCheck.TestMandatoryFields(Rec);
                end;
            }
        }
    }
}

```

```

        }
    }
    addlast(Category_Process)
    {
        actionref(MNBCheckMandatoryFields_Promoted;
MNBCheckMandatoryFields)
        {
        }
    }
}

```

CHAPTER SUMMARY

- ✓ In this chapter, you learn about the Variant, RecordRef, and FieldRef data types in AL
- ✓ You developed the functionality that can be reused for any table in the system without modifying the codebase

CHAPTER 3

LIST AND DICTIONARY

OBJECTIVES

The List and Dictionary Data Types can help you gather data to process them later in the process. The main objectives for this chapter are:

- ✓ Learn how to use lists and dictionaries
- ✓ Use both data types in the AL
- ✓ See the difference between a list and a dictionary

LIST DATA TYPE

The List data type, as the name suggests, contains a list of values. You can compare the list to the one-column table. The important information is that the type of the list needs to be specified. For example, a **List of Integer** will contain only integers and cannot contain other types. Not all types are supported in the lists – only simple types of variables like text, code, integer, decimal, etc.

The List data type is good for storing the data that needs to be later processed or to get data that is unique. The List contains useful methods that can be used to get or store unbound data.

Method	Description
Add	Allows to add the new value as last in the list. If such a value exists in the List already there will be an error.
AddRange	Allows to add the new values to the list in bulk
Contains	Check if the value already exists in the list
Count	Returns the number of elements in the list
Remove	Removes the value from the list

The list data type works perfectly with Foreach where you can iterate through all values.



LIST EXAMPLE

Below example of how to Add value to the list and how to then iterate through all values.

```
local procedure ExampleOfList()
var
    ListOfCountries: List of [Code[10]];
    Country: Code[10];
    CountryMsg: Label 'Country: %1', Comment = '%1 - Added country';
begin
    ListOfCountries.Add('US');
    ListOfCountries.Add('UK');
    ListOfCountries.Add('PL');

    foreach Country in ListOfcountries do
        Message(CountryMsg, Country);
    end;
```



TASK: SHOWING ITEMS THAT HAVE DIFFERENT UNITS OF MEASURES

The customer asked for the functionality that will filter the Item List to show only items that have different Sales or Purchase Units of Measure than the Base Unit of Measure.

1. Create a new file with a codeunit and create a new function in it **GetListOfItemsToShow** that returns the text
2. The function should find all items and check if the item has a different Base Unit Of Measure than the Sales or Purchase Unit of Measure. If so the Item should be added to the list



HINT

In this case, you do not need to check if the list contains the item since you gather it in the list Primary Key field.

3. The function should return text that contains a filter that can be applied to **No.** field on the Item table
4. Create a new file **ItemList.PagExt.al** and add page extension for **Item List**
5. Add the action that adds a filter to the list based on the return from the function created and if the text is empty show the error



SOLUTION

```
codeunit 50109 "MNB Unit Of Measure Check"
{
    internal procedure GetListOfItemsToShow(): Text
    var
        Item: Record Item;
        ListOfItems: List of [Code[20]];
        ItemNo: Code[20];
        ItemFilter: TextBuilder;

    begin
        Item.SetLoadFields("No.", "Base Unit of Measure", "Sales Unit of Measure", "Purch. Unit of Measure");
        if Item.FindSet() then
            repeat
```

```

        if (Item."Base Unit of Measure" <> Item."Sales Unit of Measure") or (Item."Base Unit of Measure" <> Item."Purch. Unit of Measure") then
            ListOfItems.Add(Item."No.");
            until Item.Next() = 0;
            if ListOfItems.Count = 0 then
                exit('');
            foreach ItemNo in ListOfItems do
                ItemFilter.Append(ItemNo + '|');
            exit(ItemFilter.ToText().TrimEnd('|'));
        end;
    }

```

```

pageextension 50103 "MNB Item List" extends "Item List"
{
    actions
    {
        addlast(Functions)
        {
            action(MNBGetListOfDifferentUnitOfMeasures)
            {
                ApplicationArea = All;
                Caption = 'Show Items with Mixed UOM';
                Image = List;
                ToolTip = 'Shows items where sales or purchase unit of measure is different from the base unit of measure.';

                trigger OnAction()
                var
                    UnitOfMeasureCheck: Codeunit "MNB Unit Of Measure Check";

                    ItemFilter: Text;
                    NoItemsFoundErr: Label 'No items found with different unit of measures.';
                begin
                    ItemFilter := UnitOfMeasureCheck.GetListOfItemsToShow();
                    if ItemFilter <> '' then
                        Rec.SetFilter("No.", ItemFilter)
                    else
                        Error(NoItemsFoundErr);
                end;
            }
        }
    }
}

```



```

        addlast(Category_Process)
    {
        actionref(MNBGetListOfDifferentUnitOfMeasures_Promoted;
MNBGetListOfDifferentUnitOfMeasures)
    {
    }
    }
}

```



TASK: COUNTING THE NUMBER OF CUSTOMERS WITH ORDERS IN STATUS OPEN

The customer asked for the quick action that shows a message on the Sales Order List to see how many customers there are open sales orders (in open status). It means one customer should not be counted twice for this information.

1. Create a new file with a codeunit and create a new function in it
CountCustomerOnSalesOrder that returns the integer.
2. The function should iterate through all Sales Orders and add to the list variable all Sell-To Customer No. that is unique.
3. The function should return the number of elements in the list.
4. Create a new Page Extension for the Sales Order List and add a new action that displays the number of unique customers.



SOLUTION

```

codeunit 50110 "MNB Sales Order Customers"
{
    internal procedure CountCustomerOnSalesOrder(): Integer
    var
        SalesHeader: Record "Sales Header";
        SalesOrderCustomerList: List of [Code[20]];
    begin
        SalesHeader.SetLoadFields("Document Type", "Sell-to Customer
No.");
    end
}

```

```

        SalesHeader.SetRange("Document Type", SalesHeader."Document
Type"::Order);
        SalesHeader.SetRange(Status, SalesHeader.Status::Open);
        if SalesHeader.FindSet() then
            repeat
                if not SalesOrderCustomerList.Contains(SalesHeader."Sell-
to Customer No.") then
                    SalesOrderCustomerList.Add(SalesHeader."Sell-to
Customer No.");
                until SalesHeader.Next() = 0;

            exit(SalesOrderCustomerList.Count);
        end;
    }

```

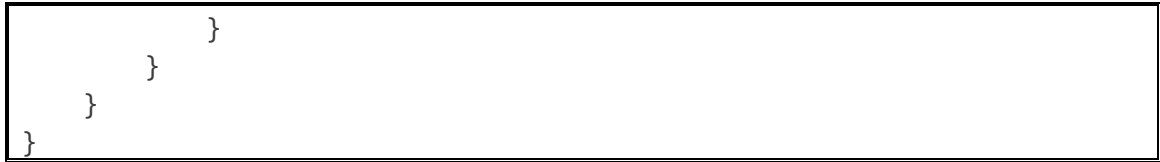
```

pageextension 50104 "MNB Sales Order List" extends "Sales Order List"
{
    actions
    {
        addlast("F&unctions")
        {
            action(MNBShowUniqueCustomerNumber)
            {
                ApplicationArea = All;
                Caption = 'Show Unique Customer Number';
                Image = Customer;
                ToolTip = 'Shows the unique customer number for all
sales orders.';

                trigger OnAction()
                var
                    SalesOrderCustomers: Codeunit "MNB Sales Order
Customers";

                    UniqueCustomerNumberMsg: Label 'The unique customer
number is %1.', Comment = '%1 - Unique customer number';
                begin
                    Message(UniqueCustomerNumberMsg,
SalesOrderCustomers.CountCustomerOnSalesOrder());
                end;
            }
        }
        addlast(Category_Process)
        {
            actionref(MNBShowUniqueCustomerNumber_Promoted;
MNBShowUniqueCustomerNumber)
            {

```



ADDITIONAL TASK: SPLIT THE TEXT INTO LINES

Try to think about how to develop functionality that will split the below text into lines that contain 50 characters or less. Remember that each line should contain the full words (words should not be split). Try to use Lists and Text functions.

Text to be split:

Business Central, a comprehensive business management solution by Microsoft, streamlines various aspects of an organization's operations. It integrates financial management, supply chain operations, sales, and customer service into a single platform, facilitating real-time data access and informed decision-making. Its user-friendly interface and seamless integration with other Microsoft products like Office 365 and Power BI enhance productivity and collaboration across departments.

DICTIONARY DATA TYPE

The Dictionary Data Type allows the creation of a collection of pair keys and values. It allows a quick search of the data collected. The key does not need to be only integer values but allows to building of dictionaries with different types of values such as text, code, date, etc. Not all data types are available to be used in the dictionary – only simple ones.



HINT

Before introducing the list and dictionary Data Type in the AL language common was to use temporary tables to collect data. Both datatypes allow us to omit that and are working much faster than using temporary tables.

The Dictionary contains useful methods that can be used to get or store data.

Method	Description
Add(Key, Value)	Allows to add the new pair to the list. If a value with the same key already exists then an error will be shown
Contains(Key)	Check if the key already exists in the dictionary
Count	Returns the number of elements in the dictionary
Get(Key)	Gets value of the key
Keys	Gets a collection containing the keys in the dictionary.
Remove(Key)	Removes the value from the dictionary
Values	Gets a collection containing the values in the dictionary.

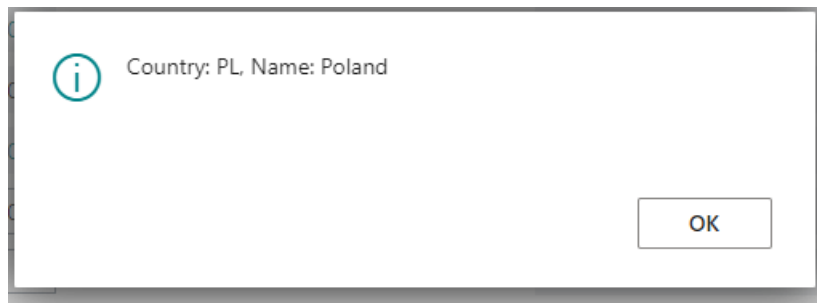


DICTIONARY EXAMPLE

Below example of how to Add value to the list and how to then iterate through all values.

```
local procedure ExampleOfDictionary()
var
    DictionaryOfcountries: Dictionary of [Code[10], Text];
    Country: Code[10];
    CountryMsg: Label 'Country: %1, Name: %2 ', Comment = '%1 - Added
country, %2 - Country name';
begin
    DictionaryOfcountries.Add('PL', 'Poland');
    DictionaryOfcountries.Add('DE', 'Germany');
    DictionaryOfcountries.Add('FR', 'France');

    foreach Country in DictionaryOfcountries.Keys do
        Message(CountryMsg, Country,
DictionaryOfcountries.Get(Country));
    end;
```



TASK: SHOWING THE NUMBER OF POSTED INVOICES AND THE REMAINING AMOUNT FOR THE NEXT 7, 14, AND 30 DAYS

The customer would like to see the information on how many posted sales invoices and what is the remaining amount for the next 7 days, 14 days, and 30 days, based on the Due Date of the invoice.

On the statistics, the customer wants only to see 3 dates and assigned values to them – 7 days, 14 days, and 30 days counted from WorkDate.

Use Dictionaries for that.

1. Create a new file with a codeunit and create a new function in it
GetPostedSalesInvoicesStatistics. As parameters use two dictionaries – one that will contain the remaining amounts and the second with a number of the invoices
2. The function should iterate through all Posted Sales Invoices and get value for specific dates for the remaining amount and number of invoices



HINT

You can first calculate what dates are from WorkDate in 7, 14, and 30 days. Later in the loop, you can add values to the proper variables. At the end, you can create dictionary keys that are the same as calculated dates and set proper values.

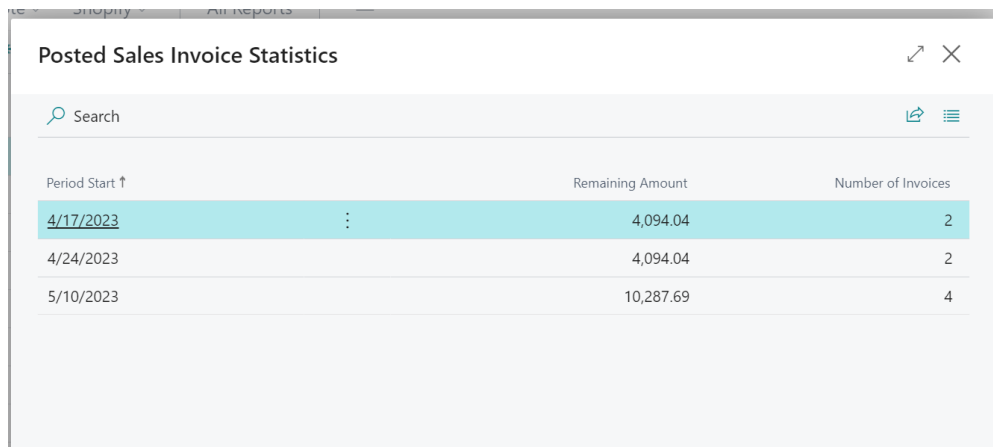
3. Create a page based on the Date table and filter values only to 3 dates that should be shown. Add 2 custom columns that will show the Remaining Amount and Number of Invoices

HINT

You can create a function in the codeunit that first will get the data, set data on the page and then run the created page.

Inside the page, you can set filters in the *OnOpenPage* trigger and values from dictionaries in the *OnAfterGetRecord* trigger

4. Created page extension for Posted Sales Invoices. Create an action to show the statistics calculated before



Period Start ↑	Remaining Amount	Number of Invoices
4/17/2023	4,094.04	2
4/24/2023	4,094.04	2
5/10/2023	10,287.69	4

SOLUTION

```
codeunit 50111 "MNB Posted Sales Invoice Stat."
{
    local procedure GetPostedSalesInvoicesStatistics(var
    RemainingAmounts: Dictionary of [Date, Decimal]; var NumberOfInvoices:
    Dictionary of [Date, Integer])
        var
            SalesInvoiceHeader: Record "Sales Invoice Header";
            RemainingAmount7Days, RemainingAmount14Days,
            RemainingAmount30Days : Decimal;
            NoOfInvoice7Days, NoOfInvoice14Days, NoOfInvoice30Days :
            Integer;
        begin
```

```

        SalesInvoiceHeader.SetLoadFields("Due Date", "Remaining
Amount");
        SalesInvoiceHeader.SetAutoCalcFields("Remaining Amount");
        if SalesInvoiceHeader.FindSet() then
            repeat
                if (SalesInvoiceHeader."Due Date" <= Date7Days) and
(SalesInvoiceHeader."Due Date" >= WorkDate()) then begin
                    RemainingAmount7Days +=
SalesInvoiceHeader."Remaining Amount";
                    NoOfInvoice7Days += 1;
                end;
                if (SalesInvoiceHeader."Due Date" <= Date14Days) and
(SalesInvoiceHeader."Due Date" >= WorkDate()) then begin
                    RemainingAmount14Days +=
SalesInvoiceHeader."Remaining Amount";
                    NoOfInvoice14Days += 1;
                end;
                if (SalesInvoiceHeader."Due Date" <= Date30Days) and
(SalesInvoiceHeader."Due Date" >= WorkDate()) then begin
                    RemainingAmount30Days +=
SalesInvoiceHeader."Remaining Amount";
                    NoOfInvoice30Days += 1;
                end;
            until SalesInvoiceHeader.Next() = 0;

        Clear(RemainingAmounts);
        Clear(NumberOfInvoices);
        RemainingAmounts.Add(Date7Days, RemainingAmount7Days);
        RemainingAmounts.Add(Date14Days, RemainingAmount14Days);
        RemainingAmounts.Add(Date30Days, RemainingAmount30Days);
        NumberOfInvoices.Add(Date7Days, NoOfInvoice7Days);
        NumberOfInvoices.Add(Date14Days, NoOfInvoice14Days);
        NumberOfInvoices.Add(Date30Days, NoOfInvoice30Days);
    end;

    local procedure GetDates()
    begin
        Date7Days := CalcDate('<7D>', WorkDate());
        Date14Days := CalcDate('<14D>', WorkDate());
        Date30Days := CalcDate('<30D>', WorkDate());
    end;

    internal procedure ShowPostedInvoiceStatistics()
    var
        PostedSalesInvStat: Page "MNB Posted Sales Inv. Stat.";
        RemainingAmounts: Dictionary of [Date, Decimal];
        NumberOfInvoices: Dictionary of [Date, Integer];
    begin

```

```

        GetDates();
        GetPostedSalesInvoicesStatistics(RemainingAmounts,
NumberOfInvoices);

        PostedSalesInvStat.GetData(RemainingAmounts, NumberOfInvoices);
        PostedSalesInvStat.RunModal();
    end;

    var
        Date7Days, Date14Days, Date30Days : Date;
}

```

```

page 50100 "MNB Posted Sales Inv. Stat."
{
    PageType = List;
    SourceTable = Date;
    ApplicationArea = All;
    Caption = 'Posted Sales Invoice Statistics';
    UsageCategory = None;
    Editable = false;

    layout
    {
        area(content)
        {
            repeater(General)
            {
                field("Date"; Rec."Period Start")
                {
                    ToolTip = 'Specifies the date.';
                }
                field(RemainingAmountField; RemainingAmount)
                {
                    ToolTip = 'Specifies the remaining amount for
specific date.';
                    Caption = 'Remaining Amount';
                }
                field(NumberOfInvoicesField; NoOfInvoice)
                {
                    ToolTip = 'Specifies the number of invoices.';
                    Caption = 'Number of Invoices';
                }
            }
        }
    }
}
var

```



```

    RemainingAmount: Decimal;
    NoOfInvoice: Integer;
    Date7Days, Date14Days, Date30Days : Date;

    RemainingAmounts: Dictionary of [Date, Decimal];
    NumberOfInvoices: Dictionary of [Date, Integer];

    trigger OnOpenPage()
    begin
        Rec.FilterGroup(9);
        Rec.SetRange("Period Type", Rec."Period Type"::Date);
        Rec.SetFilter("Period Start", '%1|%2|%3', Date7Days, Date14Days,
Date30Days);
        Rec.FilterGroup(0);
    end;

    trigger OnAfterGetRecord()
    begin
        RemainingAmount := RemainingAmounts.Get(Rec."Period Start");
        NoOfInvoice := NumberOfInvoices.Get(Rec."Period Start");
    end;

    internal procedure GetData(var RemainingAmountsLocal: Dictionary of
[Date, Decimal]; NumberOfInvoicesLocal: Dictionary of [Date, Integer])
    begin
        Date7Days := CalcDate('<7D>', WorkDate());
        Date14Days := CalcDate('<14D>', WorkDate());
        Date30Days := CalcDate('<30D>', WorkDate());
        RemainingAmounts := RemainingAmountsLocal;
        NumberOfInvoices := NumberOfInvoicesLocal;
    end;
}

```

```

pageextension 50105 "MNB Posted Sales Invoices" extends "Posted Sales
Invoices"
{
    actions
    {
        addlast(processing)
        {
            action(MNBShowInvoiceStats)
            {
                ApplicationArea = All;
                Caption = 'Show Invoice Stats';
            }
        }
    }
}

```

```

        Image = Statistics;
        ToolTip = 'Shows the statistics based on the due date
and remaining quantity.';

        trigger OnAction()
        var
            PostedSalesInvoiceStat: Codeunit "MNB Posted Sales
Invoice Stat.";
        begin
            PostedSalesInvoiceStat.ShowPostedInvoiceStatistics()
;
        end;
    }
}
addlast(Category_Process)
{
    actionref(MNBShowInvoiceStats_Promoted; MNBShowInvoiceStats)
    {
    }
}
}
}

```



ADDITIONAL TASK: IMPROVE THE ABOVE SOLUTION

The code above is just an example of using the dictionaries and definitely can be improved. You may think about how to do so that the solution will require almost no additional development if the customer would like to show additionally to already presented data also 60, 90, 120, and 365 days from the WorkDate. You should think of using List and Foreach for that.

CHAPTER SUMMARY

- ✓ In this chapter, you learn about the List and Dictionary data types.
- ✓ You developed functionalities that required to use of those types.

CHAPTER 4

HANDLING ERROR IN AL

OBJECTIVES

The error messages in Business Central should tell more users than just an error. The message should allow to unblock the user when required. Also can give more information than just a message. The main objectives for this part of the workbook are:

- ✓ Learn how to build better error messages
- ✓ How to add actions for the messages
- ✓ Learn how to collect the messages to show all errors at the same time

INTRODUCTION

In the AL language, there are more options to handle the error messages than just **Error()**. Depending on the usage and the user experience it is possible to choose in the example below ways to handle the errors:

- ✓ **TryFunction** – when added to the procedure it will return true or false if the code was run successfully or not
- ✓ **Run Codeunit** – running the Codeunit in **if... then** statement, similar to TryFunction, will return the true or false if the code has been executed. In the past, it was the only way to get a similar behavior as the try function.
- ✓ **ErrorBehavior** – when added to the procedure it allows collecting the error messages and shows the user all errors at the same time

ERRORINFO DATA TYPE

For a better user experience, it is possible to add not only error text to the Error but also a title or action that can give the user the possibility to unblock the process – for example pointing the user to the proper setup page or even setting the new value of the field.

For that, **ErrorInfo** can be used which is later added to the Error(). It comes with useful methods. Some of them are shown in the below table.

Method	Description
Create	Creates an instance of the ErrorInfo
Title	Specifies the title of the error shown to the user
Message	Specifies the message shown to the user
RecordId	Specifies the record ID to which the error message belongs to
FieldId	Specifies the field ID to which the error message belongs to
SystemId	Specifies the system Id which the error message belongs to
AddAction	Adds action that will run the specific procedure in the codeunit that has been pointed to be triggered

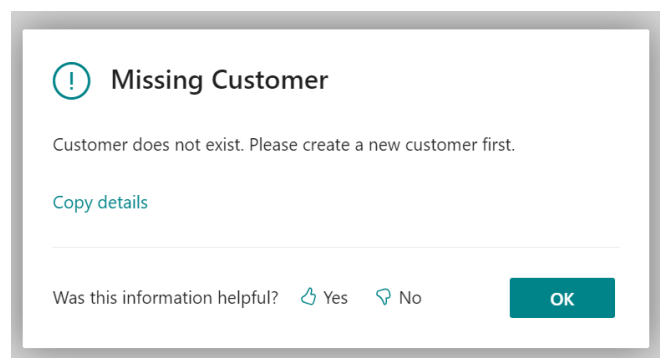
CustomDimensions	Allows to add additional dimension to the error message. It would be useful in the action that will allow fixing error
AddNavigationAction	Adds action that will open a specific page
PageNo	Defines which page should be open when clicking Navigation Action



EXAMPLE ERRORINFO – SIMPLE ERROR MESSAGE

The below example shows the error message with the title and message. It also adds a detailed message that will be visible when copying the details of the message.

```
local procedure SimpleErrorInfo()
var
    ErrorInfoMsg: ErrorInfo;
    CustomerIsMissingMsg: Label 'Missing Customer';
    CustomerDoesNotExistErr: Label 'Customer does not exist. Please
create a new customer first.';
    DetailedCustomerDoesNotExistErr: Label 'User tried to create a sales
invoice for the customer, but the customer does not exist in the
system.';
begin
    ErrorInfoMsg := ErrorInfo.Create(CustomerDoesNotExistErr, true);
    ErrorInfoMsg.Title(CustomerIsMissingMsg);
    ErrorInfoMsg.DetailedMessage(DetailedCustomerDoesNotExistErr);
    Error(ErrorInfoMsg);
end;
```

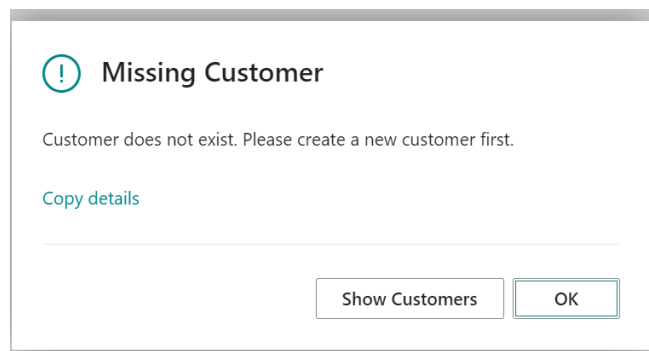




EXAMPLE ERRORINFO –ERROR MESSAGE WITH NAVIGATION

The below example shows the error message that contains the navigation action that can help unblock the user.

```
local procedure NavigationActionErrorInfo()  
var  
    ErrorInfoMsg: ErrorInfo;  
    CustomerIsMissingMsg: Label 'Missing Customer';  
    CustomerDoesNotExistErr: Label 'Customer does not exist. Please  
create a new customer first.';  
    DetailedCustomerDoesNotExistErr: Label 'User tried to create a sales  
invoice for the customer, but the customer does not exist in the  
system.';  
    OpenCustomerListTxt: Label 'Show Customers';  
begin  
    ErrorInfoMsg := ErrorInfo.Create(CustomerDoesNotExistErr, true);  
    ErrorInfoMsg.Title(CustomerIsMissingMsg);  
    ErrorInfoMsg.DetailedMessage(DetailedCustomerDoesNotExistErr);  
    ErrorInfoMsg.AddNavigationAction(OpenCustomerListTxt);  
    ErrorInfoMsg.PageNo(Page::"Customer List");  
    Error(ErrorInfoMsg);  
end;
```



EXAMPLE ERRORINFO –ERROR MESSAGE WITH CUSTOM ACTION

The below example shows the error message that contains the custom action that will run the codeunit "MNB Error Actions" and function **CreateCustomer**. The procedure has in the parameters the ErrorInfo data type that contains all information about the message such as SystemId if set or custom dimensions.

```

codeunit 50112 "MNB Error Actions"
{
    SingleInstance = true;
    procedure CreateCustomer(SentErrorInfo: ErrorInfo)
    var
        Customer: Record Customer;
        CustomerName: Text[100];
    begin
        CustomerName :=
SentErrorInfo.CustomDimensions.Get('CustomerName');

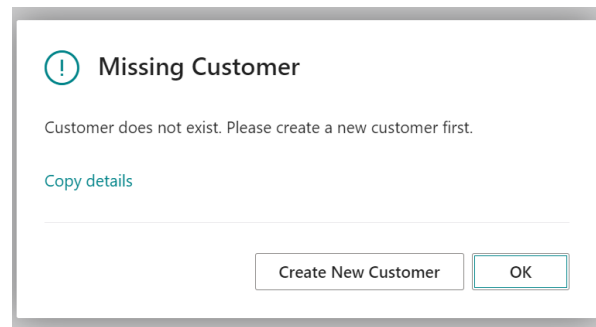
        Customer.Init();
        Customer.Name := CustomerName;
        Customer.Insert(true);
    end;
}

```

```

local procedure FixActionErrorInfo()
var
    ErrorInfoMsg: ErrorInfo;
    CustomerIsMissingMsg: Label 'Missing Customer';
    CustomerDoesNotExistErr: Label 'Customer does not exist. Please
create a new customer first.';
    DetailedCustomerDoesNotExistErr: Label 'User tried to create a sales
invoice for the customer, but the customer does not exist in the
system.';
    CreateCustomerTxt: Label 'Create New Customer';
begin
    ErrorInfoMsg := ErrorInfo.Create(CustomerDoesNotExistErr, true);
    ErrorInfoMsg.Title(CustomerIsMissingMsg);
    ErrorInfoMsg.DetailedMessage(DetailedCustomerDoesNotExistErr);
    ErrorInfoMsg.CustomDimensions.Add('CustomerName', 'New Corporation
Customer');
    ErrorInfoMsg.AddAction(CreateCustomerTxt, Codeunit::"MNB Error
Actions", 'CreateCustomer');
    ErrorInfoMsg.RecordId(Rec.RecordId);
    ErrorInfoMsg.PageNo(Page::"Customer Card");
    Error(ErrorInfoMsg);
end;

```



You can see the detailed information and custom dimensions sent with an error when copying details to the clipboard.

```
If requesting support, please provide the following details to help troubleshooting:
Customer does not exist. Please create a new customer first.
User tried to create a sales invoice for the customer, but the customer does not exist in the system.
Internal session ID:
e06ee29a-597a-477d-9bef-c200772fc0c6
Application Insights session ID:
ed9dc835-c72e-4529-8f14-e6a81a5ee492
Client activity id:
7877557b-ae49-48ba-830d-dcd43f741bd8
Time stamp on error:
2024-02-11T16:02:10.9022172Z
User telemetry id:
12613c65-6d71-4d87-89c9-c132d08bf902
AL call stack:
"MNB Error Handling Examples"(Page 50102).FixActionErrorInfo line 15 - Advanced Development Workshop by myNAVblog.com
"MNB Error Handling Examples"(Page 50102).CheckCustomer - OnAction"(Trigger) line 2 - Advanced Development Workshop by myNAVblog.com
Custom dimensions:
[{"Item1":"CustomerName","Item2":"New Corporation Customer"}]
```

When developing errors follow the guidance about how to write good error messages from Microsoft that can be found here: <https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-error-handling-guidelines>

COLLECTING ERRORS

Instead of showing the user one error at a time in the process, it is possible to collect all errors and show them on one page. To do so it is possible to add how errors are collected. To do that it is needed to declare before function **[ErrorBehavior(ErrorBehavior::Collect)]**. Additionally, the ErrorInfo needs to be set as collectible.

When collecting errors two statements can be helpful:

- ✓ **HasCollectedErrors** – returns true if there are any errors collected
- ✓ **system.GetCollectedErrors()** – allows to get collected errors in the foreach loop

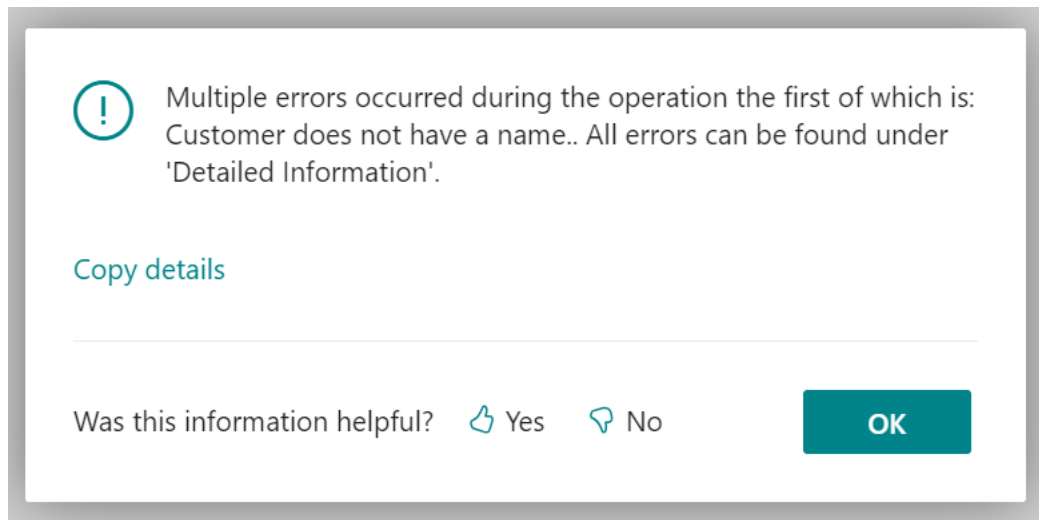


EXAMPLE COLLECT – MULTIPLE ERRORS IN ONE ERROR MESSAGE

The below example shows a message that multiple errors occurred during the transaction. The second parameter in the **Create** method for **ErrorInfo** sets it as collectible.

```
[ErrorBehavior(ErrorBehavior::Collect)]
procedure MultipleErrorCollect()
var
    ErrorInfoMsg, ErrorInfoMsg2 : ErrorInfo;
    CustomerDoesNotHaveNameErr: Label 'Customer does not have a name.';
    CustomerDoesNotHaveAddressErr: Label 'Customer does not have an
address.';
begin
    ErrorInfoMsg.Collectible(true);
    ErrorInfoMsg.Message(CustomerDoesNotHaveNameErr);
    Error(ErrorInfoMsg);

    ErrorInfoMsg2.Collectible(true);
    ErrorInfoMsg2.Message(CustomerDoesNotHaveAddressErr);
    Error(ErrorInfoMsg2);
end;
```



If requesting support, please provide the following details to help troubleshooting:

Multiple errors occurred during the operation the first of which is: Customer does not have a name.. All errors can be found under 'Detailed Information'.

Customer does not have a name.
Customer does not have an address.

Internal session ID:
ac4c4535-080f-4652-939f-f5ad6410b616

Application Insights session ID:
288c64fd-3e8d-47bd-a1a0-385019faf5bc

Client activity id:
9908072e-711e-4177-ba09-f27740a71b0c

Time stamp on error: |
2024-02-11T21:20:15.0285375Z

User telemetry id:
12613c65-6d71-4d87-89c9-c132d08bf902



EXAMPLE COLLECT – MULTIPLE ERRORS SHOWN IN THE LIST

The below example shows how to add to the previous example list where the user can see all errors. The collected errors are populated at the end of the procedure in the temporary table "Error Message" and show the page with errors.

```
[ErrorBehavior(ErrorBehavior::Collect)]
procedure MultipleErrorCollect()
var
    TempErrorMessage: Record "Error Message" temporary;
    ErrorInfoMsg, ErrorInfoMsg2, CollectedErrorInfo : ErrorInfo;
    CustomerDoesNotHaveNameErr: Label 'Customer does not have a name.';
    CustomerDoesNotHaveAddressErr: Label 'Customer does not have an
address.';
begin
    ErrorInfoMsg.Collectible(true);
    ErrorInfoMsg.Message(CustomerDoesNotHaveNameErr);
```

```

Error(ErrorInfoMsg);

ErrorInfoMsg2.Collectible(true);
ErrorInfoMsg2.Message(CustomerDoesNotHaveAddressErr);
Error(ErrorInfoMsg2);

if HasCollectedErrors then begin
    foreach CollectedErrorInfo in system.GetCollectedErrors() do
        TempErrorMessage.LogSimpleMessage(0,
CollectedErrorInfo.Message);
        ClearCollectedErrors();

        TempErrorMessage.ShowErrorMessage(true);
end;
end;

```

Description	Recommended action	Message Type	Status
<u>Customer does not have a name.</u>	-	Error	
Customer does not have an address.	-	Error	

HINT

It is possible to define if the record is an error, warning, or information.

In the table "Error Message" there are also different procedures to Log the message. For example,

***LogDetailedMessage** also gives the possibility to send information about the record and more details with a help link as well.*

Make sure to set the parameter to true in ShowErrorMessage, otherwise the transaction is not rolled back.

IGNORING COMMIT WHEN COLLECTING ERRORS

The issue that you may face while collecting errors in one transaction and showing them on the page is the situation where in the code the Commit has been added. Then the transaction will not be rolled back. To avoid issues with it you can add before procedure information on how to commit should be treated: **[CommitBehavior(CommitBehavior::Ignore)]**.

Try to run the below code with and without information on how the commit should be treated. You should get two different results.

```
[ErrorBehavior(ErrorBehavior::Collect)]
[CommitBehavior(CommitBehavior::Ignore)]
procedure MultipleErrorCollect()
var
    Customer: Record Customer;
    TempErrorMessage: Record "Error Message" temporary;
    ErrorInfoMsg, ErrorInfoMsg2, CollectedErrorInfo : ErrorInfo;
    CustomerDoesNotHaveNameErr: Label 'Customer does not have a name.';
    CustomerDoesNotHaveAddressErr: Label 'Customer does not have an
address.';
begin
    ErrorInfoMsg.Collectible(true);
    ErrorInfoMsg.Message(CustomerDoesNotHaveNameErr);
    Error(ErrorInfoMsg);

    ErrorInfoMsg2.Collectible(true);
    ErrorInfoMsg2.Message(CustomerDoesNotHaveAddressErr);
    Error(ErrorInfoMsg2);

    Customer.FindFirst();
    Customer.Name := 'New Corporation Customer';
    Customer.Modify(true);
    Commit();

    if HasCollectedErrors then begin
        foreach CollectedErrorInfo in system.GetCollectedErrors() do
            TempErrorMessage.LogSimpleMessage(0,
CollectedErrorInfo.Message);
            ClearCollectedErrors();

            TempErrorMessage.ShowErrorMessage(true);
        end;
    end;
```

HINT

The information about the behavior of commit can be useful when using TryFunction

TASK: SHOWING THE BETTER ERROR MESSAGE

The customer asked for the functionality that will show an error message when anyone tries to set Blocked All on the Customer Card when there are any open sales orders for the customer.

Remember to make the error understandable for the user and add also action to open a list of Sales Orders.

1. Create a new file for table extension for the **Customer** table and add code before validating the **Blocked** field
2. Add `ErrorMessage` data type as a variable and proper labels
3. In the code check if the **Blocked** field value is **All** and if for Customer are any sales orders
4. If yes then prepare `ErrorMessage`. Remember to first create it, add a message, title, and navigation action that will open the page **"Sales Order List"**. Make sure that the title and message are meaningful for the user



SOLUTION

```
tableextension 50101 "MNB Customer" extends Customer
{
    fields
    {
        modify(Blocked)
        {
            trigger OnBeforeValidate()
            begin
                CheckOpenSalesOrders();
            end;
        }
    }

    local procedure CheckOpenSalesOrders()
    var
        ErrorInfoMsg: ErrorInfo;
        CustomerCannotHaveOrdersErr: Label 'Customer has open orders';
        OrderErrorWhenBlockingErr: Label 'When blocking the customer,
the customer cannot have open orders. Please close all open orders
before blocking the customer.';
        ShowOrdersLbl: Label 'Show orders';
    begin
        if Rec.Blocked <> Blocked::All then
            exit;
        Rec.CalcFields("No. of Orders");
        if Rec."No. of Orders" = 0 then
            exit;
        end if;
    end
}
```

```



        ErrorInfoMsg := ErrorInfo.Create(OrderErrorWhenBlockingErr,
true);
        ErrorInfoMsg.Title(CustomerCannotHaveOrdersErr);
        ErrorInfoMsg.PageNo(Page::"Sales Order List");
        ErrorInfoMsg.AddNavigationAction>ShowOrdersLbl);
        Error(ErrorInfoMsg);
    end;
}

```

Balance Due (\$)	0.00	Profit
Credit Limit (\$)	0.00	Profit
Blocked	All	Last I
Privacy Blocked		isab
Salesperson Code		equ

Customer has open orders.

When blocking the customer, the customer cannot have open orders. Please close all open orders before blocking the customer.

Show orders  



TASK: OPEN SALES ORDERS ONLY FOR SPECIFIC CUSTOMER

The customer is happy with the solution that has been provided. However, they would like that the list of sales orders will contain only sales orders for specific customers. Use the action for the Error Info instead of the navigation action.

1. Create a new codeunit "**MNB Actions Error - Customer**" and procedure **OpenSalesOrders** that has in the parameters ErrorInfo data type
2. In the procedure get the value of the custom dimension with key **CustomerNo** and then show the Sales Order List filtered for specific customer
3. Open the table extension from the previous task and change the existing procedure. Instead of **AddNavigationAction** use **AddAction** to add also Custom Dimension **CustomerNo**



SOLUTION

```

codeunit 50113 "MNB Actions Error - Customer"
{
    procedure OpenSalesOrders(ErrorInfoMsg: ErrorInfo)

```

```

var
    SalesHeader: Record "Sales Header";
    CustomerNo: Code[20];
begin
    CustomerNo := ErrorInfoMsg.CustomDimensions.Get('CustomerNo');
    if CustomerNo = '' then
        exit;
    SalesHeader.SetRange("Document Type", SalesHeader."Document
Type"::Order);
    SalesHeader.SetRange("Sell-to Customer No.", CustomerNo);
    Page.RunModal(Page::"Sales Order List", SalesHeader);
end;
}

```

```

local procedure CheckOpenSalesOrdersForCustomer()
var
    ErrorInfoMsg: ErrorInfo;
    CustomerCannotHaveOrdersErr: Label 'Customer has open orders.';
    OrderErrorWhenBlockingErr: Label 'When blocking the customer, the
customer cannot have open orders. Please close all open orders before
blocking the customer.';
    ShowOrdersLbl: Label 'Show orders';
begin
    if Rec.Blocked <> Blocked::All then
        exit;
    Rec.CalcFields("No. of Orders");
    if Rec."No. of Orders" = 0 then
        exit;
    ErrorInfoMsg := ErrorInfo.Create(OrderErrorWhenBlockingErr, true);
    ErrorInfoMsg.Title(CustomerCannotHaveOrdersErr);
    ErrorInfoMsg.CustomDimensions.Add('CustomerNo', Rec."No.");
    ErrorInfoMsg.AddAction(ShowOrdersLbl, Codeunit::"MNB Actions Error -
Customer", 'OpenSalesOrders');
    Error(ErrorInfoMsg);
end;

```



TASK: MANDATORY FIELDS – ONE LIST WITH ALL ERRORS

The customer has been using the functionality developed before related to Mandatory Fields

however would like to see all errors in one list instead of fixing each field one by one. Use collecting errors for that.

1. Open the codeunit "MNB Mandatory Field Check" and change the existing procedure **TestMandatoryFields** (this has been created in one of the previous tasks related to RecordRef and FieldRef).
2. Add to the procedure **ErrorBehavior**
3. Add instead of a simple error message the ErrorInfo data type and set the message, **Record Id**, and **Field No**.
4. Add checking if there are any collected errors and if so populate the temporary record in the table "Error Message"
5. After that clear collected errors and show a list of errors



SOLUTION

```
[ErrorBehavior(ErrorBehavior::Collect)]
internal procedure TestAllMandatoryFields(RecToCheck: Variant)
var
    MandatoryFieldsSetup: Record "MNB Mandatory Field Setup";
    TempErrorMessage: Record "Error Message" temporary;
    RecRef: RecordRef;
    FldRec: FieldRef;
    ErrorInfoMsg, CollectedErrorInfo : ErrorInfo;
    FieldIsMandatoryErr: Label 'Field %1 is mandatory.', Comment = '%1 -
field caption';
    FieldsAreSetMsg: Label 'All mandatory fields for %1 are filled in.',
Comment = '%1 - table caption';
begin
    if not RecToCheck.IsRecord then
        exit;

    RecRef.GetTable(RecToCheck);
    MandatoryFieldsSetup.SetRange("Table No.", RecRef.Number);
    if MandatoryFieldsSetup.FindSet() then begin
        repeat
            FldRec := RecRef.Field(MandatoryFieldsSetup."Field No.");
            if Format(FldRec.Value) = '' then begin
                ErrorInfoMsg :=
ErrorInfo.Create(StrSubstNo(FieldIsMandatoryErr, FldRec.Caption), true);
                ErrorInfoMsg.RecordId(RecRef.RecordId);
                ErrorInfoMsg.FieldNo(FldRec.Number);
                Error(ErrorInfoMsg);
            end;
        until FldRec.Next() = 0;
    end;
end;
```



```

until MandatoryFieldsSetup.Next() = 0;
if not HasCollectedErrors then
    Message(FieldsAreSetMsg, RecRef.Caption)
else begin
    foreach CollectedErrorInfo in system.GetCollectedErrors() do
        TempErrorMessage.LogSimpleMessage(0,
CollectedErrorInfo.Message);

        ClearCollectedErrors();
        TempErrorMessage.ShowErrorMessage(true);
    end;
end;
end;

```

← Error Messages 🔗 ↻

🔍 Search ☒ Analyze ☒ Accept recommended action ☒ Hide fixed errors ... 🔗 🔍 ☰ ⓘ

Description	Recommended action	Message Type	Status
<u>Field Name is mandatory.</u>	⋮ —	Error	
Field Address is mandatory.	—	Error	
Field Address 2 is mandatory.	—	Error	

CHAPTER SUMMARY

- ✓ In this chapter, you learned how to create more userfriendly error messages than just simple Error
- ✓ You also learned how to collect errors on one page so the user is informed about all issues at once
- ✓ You developed functionalities that show the errors with titles, and actions and also that collect errors

CHAPTER 5

QUERIES

OBJECTIVES

Queries can be a very powerful tool to collect and later show data inside Business Central from multiple tables at the same time. In this chapter goal is to:

- ✓ Learn how to build queries
- ✓ How to show queries inside Business Central
- ✓ How to read data from the query to build more advanced calculations

QUERY OBJECT

The Query object is used to get a set of data from one or more tables. It is also possible to aggregate the data retrieved with the query, for example, to get the sum of records.

Queries in Business Central can be used to

- ✓ present data to the user
- ✓ used as a source of custom APIs (please see for examples chapter Custom API)
- ✓ also can be used in AL for some operations like gathering data and processing it later

QUERY PROPERTIES

The query properties are added for the whole object. Below are just basic properties that are needed to create a query.

QueryType	For queries that are not exposed through API query type is Normal.
Query	
Caption	Caption for the query. It should not contain the prefix or suffix.
UsageCategory	Specifies if the query should be visible from the Tell Me functionality

QUERY ELEMENTS

Similar to the report object, to Query it is possible to add such elements as **dataitem**, **column**, and additionally also **filter** to the query. The top-level always needs to be **dataitem** that represents the table from which the records should be taken.

DATAITEM

For the **dataitem**, there are below properties that can be useful

DataItemLink	Allows to connect the dataitems between each other.
DataItemTableFilter	Allows to set filters for the data items.
SqlJoinType	Allows to specify type between data items in a query to determine the records that are included in the resulting data set.

HINT

You cannot combine two DataItems at the same level because unions are not supported.

*For more information on options for **SqlJoinType** read here: https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/properties/devenv-sqljointype-property?wt.mc_id=d365bc_inproduct_alextension*

COLUMN

For the column, the element shows the specific table column values or calculated values based on the aggregation.

Columns also have parameters that can be useful during the development.

Caption	Allows to set the custom caption of the column. Can be needed when presenting data to the user.
ColumnFilter	Allows to add filters for the specific column. And can be overwritten from the code. If the ColumnFilter and DataItemTableFilter are specified for the same field then the SQL statement is translated to AND.
Method	Allows to aggregate data as like: Count, Sum, Average, Max, or Min.

FILTER

The filter element in the query allows to addition of additional filters for the specific field in the dataitem. However, the filter element is not displayed in the UI nor is possible to retrieve data via code.

QUERY TRIGGERS

There is only one trigger for the query object. The **OnBeforeOpen()** is triggered before opening the query.

VIEWING QUERIES FROM BUSINESS CENTRAL

Similar to other object types you can create an action to run an object with a type query.

```
action(MNBMyItems)
{
    ApplicationArea = All;
    Image = Report2;
    Caption = 'My Items';
    ToolTip = 'Shows the open sales orders by country.';
    RunObject = query "My Items";
}
```



TASK: OPEN SALES ORDERS BY COUNTRY, REGION AND CITY

The customer asked you to create the report for the users that will show from the Countries page.

The users would like to see Country Code, Name, State (Region), City, Amount in Sales Orders, and Number of Sales Orders.

It turns out that users don't want to print the report and don't need also to export it to Excel – they would like to see it on the screen.

1. Create a new file and create a new query object using snippet **tquery**
2. Add a Caption to the query object. You may also consider adding other properties like **AboutText** and **AboutTitle**
3. Add dataitem related to the **Country/Region** table and add columns **Code** and **Name**
4. Add new dataitem related to the **Sales Header** and add proper **DataItemLink** and proper **DataItemTableFilter**

5. Add columns **Sell-to County, Sell-to City**
6. Add column **Amount** and add Method property to sum value from all records
7. Add column **NumberOfSalesOrders** that is not based on any field and add method Count to calculate the number of records
8. Create a new page extension for the Countries/Regions page and add the new action to run the query created



SOLUTION

```
query 50100 "MNB Sales By Country/City"
{
    QueryType = Normal;
    Caption = 'Sales By Country/City';

    elements
    {
        dataitem(Country_Region; "Country/Region")
        {
            column(Code; Code) { }
            column(Name; Name) { }
            dataitem(Sales_Header; "Sales Header")
            {
                DataItemLink = "Sell-to Country/Region Code" =
Country_Region.Code;
                DataItemTableFilter = "Document Type" = filter(Order);
                column(Sell_to_County; "Sell-to County")
                {
                }
                column(Sell_to_City; "Sell-to City")
                {
                }
                column(Amount; Amount)
                {
                    Method = Sum;
                }
                column(NumberOfSalesOrders)
                {
                    Caption = 'Number of Sales Orders';
                    Method = Count;
                }
            }
        }
    }
}
```

```

    }
  }
}

```

```

pageextension 50106 "MNB Countries/Regions" extends "Countries/Regions"
{
    actions
    {
        addlast(Reporting)
        {
            action(MNBSalesByCountryCity)
            {
                ApplicationArea = All;
                Image = Report2;
                Caption = 'Sales By Country/City';
                ToolTip = 'Shows the open sales orders by country.';
                RunObject = query "MNB Sales By Country/City";
            }
        }
        addlast(Category_Report)
        {
            actionref(MNBSalesByCountryCity_Promoted;
MNBSalesByCountryCity)
            {
            }
        }
    }
}

```

Sales By Country/City

Code	Name	Sell-to State	Sell-to City	↓ Numbe.	Amount	Pivot Mode
US	USA	GA	Atlanta	30	57,209.95	<input checked="" type="checkbox"/>
US	USA	FL	Miami	3	33,479.15	<input checked="" type="checkbox"/>
US	USA	IL	Chicago	2	360.00	<input checked="" type="checkbox"/>

Search...

☒ Code
 ☒ Name
 ☒ Sell-to State
 ☒ Sell-to City
 ☒ Number of Sales Orders
 ☒ Amount

Row Groups

Drag here to set row groups

Values

☒ Sum(Number of Sal...)
 ☒ Sum(Amount)

USING QUERIES IN AL

The query object can be used directly in AL to read the data defined in the query. In some cases, it is more convenient and performance-wise to use queries instead of other methods. Remember that it is not possible to modify data using the query. To get data using query you need to do operations in this order:

- ✓ Add filters to query using SetRange or SetFilter
- ✓ Run method Open() on the query to get the records
- ✓ Add loop while ... Read() to be able to read each record in the result
- ✓ Run method Close() on the query to return the query instance to the initialized state



TASK: INFORMATION ABOUT SALES FOR COUNTRY

The customer asked you to create the message to show the Total Amount, Total Remaining Amount, Total Sales Amount Current Month, and Total Remaining Amount Current Month for the specific county.

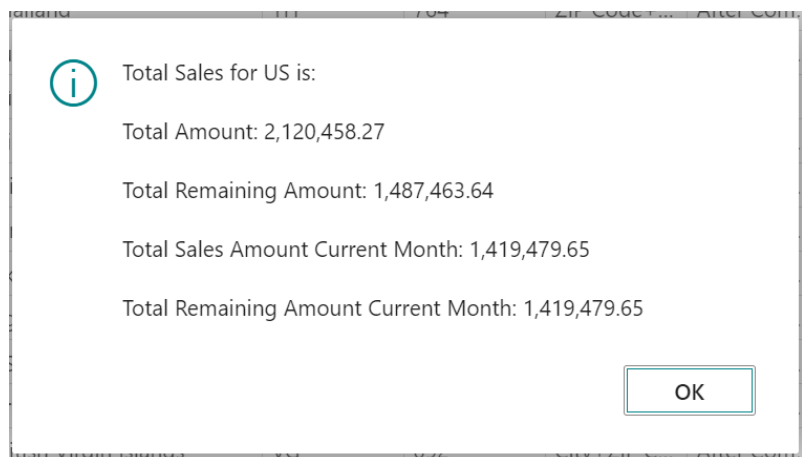
Use a query that will gather information for the message. Add it to the Countries/Regions Page.

1. Create a new file and create a new query object using snippet **tquery**
2. Add as dataitem for table **Cust. Ledger Entries** and add a proper **DataItemTableFilter**
3. Add two columns based on the **Posting Date** that show the Month and Year
4. Add columns based on **Amount** and **Remaining Amount** and make them as sums
5. Add new dataitem for table Customer and add proper **DataItemLink**
6. Add a column showing the customer's country code



You can test if the query shows the proper data running it from Business Central

7. Create a new codeunit and add a new procedure that takes as a parameter the Country Code
8. Apply proper filters to the query and open it
9. Iterate through query rows and gather data for the variables to show them later in the message
10. Close the query and show the message
11. Create a new action in the MNB Countries/Regions page extension and run the procedure from the codeunit





SOLUTION

```
query 50101 "MNB Customer Ledg. by Country"
{
    QueryType = Normal;

    elements
    {
        dataitem(Cust__Ledger_Entry; "Cust. Ledger Entry")
        {
            DataItemTableFilter = "Document Type" = const(Invoice);
            filter(Posting_Date; "Posting Date") { }
            column(Posting_Date_Month; "Posting Date")
            {
                Method = Month;
            }
            column(Posting_Date_Year; "Posting Date")
            {
                Method = Year;
            }
            column(Amount; Amount)
            {
                Method = Sum;
            }
            column(Remaining_Amount; "Remaining Amount")
            {
                Method = Sum;
            }

            dataitem(Customer; "Customer")
            {
                DataItemLink = "No." = Cust__Ledger_Entry."Customer
No.";
                column(Country_Region_Code; "Country/Region Code") { }
            }
        }
    }
}
```

```
codeunit 50114 "MNB Countries Statistics"
{
    internal procedure ShowTotalSalesForCountry(CountryCode: Code[10])
    var
        CustomerLedgByCountry: Query "MNB Customer Ledg. by Country";
```

```

        SalesMsg: Label 'Total Sales for %1 is: \\ Total Amount: %2 \\
Total Remaining Amount: %3 \\ Total Sales Amount Current Month: %4 \\
Total Remaining Amount Current Month: %5', Comment = '%1 = Country Code,
%2 = Total Sales Amount, %3 = Total Remaining Amount, %4 = Total Sales
Amount Current Year, %5 = Total Remaining Amount Current Year';
        TotalSalesAmount, TotalRemainingAmount : Decimal;
        TotalSalesAmountCurrentMonth, TotalRemainingAmountCurrentMonth :
Decimal;
        begin
            CustomerLedgByCountry.SetRange(CustomerLedgByCountry.Country_Reg
ion_Code, CountryCode);
            CustomerLedgByCountry.Open();
            while CustomerLedgByCountry.Read() do begin
                TotalSalesAmount += CustomerLedgByCountry.Amount;
                TotalRemainingAmount +=
CustomerLedgByCountry.Remaining_Amount;
                if (CustomerLedgByCountry.Posting_Date_Year =
Date2DMY(WorkDate(), 3)) and (CustomerLedgByCountry.Posting_Date_Month =
Date2DMY(WorkDate(), 2)) then begin
                    TotalSalesAmountCurrentMonth +=
CustomerLedgByCountry.Amount;
                    TotalRemainingAmountCurrentMonth +=
CustomerLedgByCountry.Remaining_Amount;
                end;
            end;
            CustomerLedgByCountry.Close();
            Message(SalesMsg, CountryCode, TotalSalesAmount,
TotalRemainingAmount, TotalSalesAmountCurrentMonth,
TotalRemainingAmountCurrentMonth);
        end;
    }

```

```

action(MNBSHowSalesForCountry)
{
    ApplicationArea = All;
    Image = Report2;
    Caption = 'Show Total Sales for Country';
    ToolTip = 'Shows the open sales orders by country.';
    trigger OnAction()
    var
        CountriesStatistics: Codeunit "MNB Countries Statistics";
    begin
        CountriesStatistics.ShowTotalSalesForCountry(Rec.Code);
    end;
}

```

CHAPTER SUMMARY

- ✓ In this chapter, you learn about the queries and how to use them
- ✓ You developed the query that presents the data directly to the user
- ✓ You also learned how to use the query directly in AL code

CHAPTER 6

CUSTOM APIs AND EXTERNAL BUSINESS EVENTS

OBJECTIVES

Custom APIs are very important for any integrations between Business Central and external systems. Also with Power Platform. In this chapter the objectives are to:

- ✓ Learn how to make custom APIs based on Pages and Queries
- ✓ Learn how to create actions that can be run from the API
- ✓ Learn how to use External Business Events
- ✓ Develop APIs and use them in the Power Platform

CUSTOM APIs

In the modern world when organizations have multiple applications APIs are one of the fastest ways to build connections between other software and Business Central. Since API integration is more just the code, In this workbook I will focus on the AL development of APIs and will not focus on the authentication part of the process.

The tasks in this chapter will require basic knowledge of Power Automate and Power BI.



To learn more information about APIs visit AJ Kauffmann's blog <https://www.kauffmann.nl> where he explains all details about APIs

Although Business Central has built-in APIs, there is often a need to prepare a set of custom APIs when building integrations. You can use APIs in example for:

- ✓ Integration with Power Automate or Logic Apps
- ✓ Integration with PowerBI
- ✓ Integration with Dataverse
- ✓ Integration with external services and applications

It is possible to expose Pages or Queries as APIs. Both types of objects have common properties that are required for APIs.

PageType, QueryType	
APIGroup	Mandatory property that specifies to which group the API belongs.
APIPublisher	Mandatory property that specifies who is the API publisher. In PTE common practice is to use customer name instead of own company name.
APIVersion	Mandatory property that defines the version of API. It can have multiple versions. Follow the pattern vX.Y for the version. Note: Beta versions are not visible in Power Automate but can be used when the version is specified as a custom value

EntityName	Sets the singular entity name with which the page is exposed in the API endpoint.
-------------------	---

EntitySetName	Sets the plural entity name with which the page is exposed in the API endpoint.
----------------------	---

DataAccessIntent	Specifies what purpose of getting data from the API. Using ReadOnly value might improve performance.
-------------------------	--

EntityCaption, EntitySetCaption	Sets the captions for Entry (singular) and EntitySet (popular)
--	--

Caption	Sets a caption for the objects
----------------	--------------------------------



HINT

All the values in the object properties and names of the fields should have camelcase names.

PAGE APIs

In additionally to the above common parameters, the API pages can have others like:

ModifyAllowed, InsertAllowed, DeleteAllowed	Specifies what operations are allowed to the record
--	---

ODataKeyFields	Specifies the key fields when using OData. In many cases, the System Id field is enough to use it.
-----------------------	--

SourceTable	Specifies the same as in standard pages what is the source table for the API
--------------------	--

DelayedInsert	This parameter for the API pages needs to be set to true
----------------------	--



HINT

When defining the fields for the page API make sure to expose the SystemID as id field.

When using No. fields use as name number instead of No.

As with standard pages, it is possible to add the subpages (parts) to the API. The subpage should have the same API version as the main page. It is also required to define the entityName, entitySetName, and SubPageLink. An example you can find below is from standard Customer API.

```

part(defaultDimensions; "APIV2 - Default Dimensions")
{
    Caption = 'Default Dimensions';
    EntityName = 'defaultDimension';
    EntitySetName = 'defaultDimensions';
    SubPageLink = ParentId = field(SystemId), "Parent Type" = const(Customer);
}

```



TASK: API FOR PROJECTS

The customer asked you to integrate between their external project system and Business Central using APIs (and Power Automate). It is needed to create the API for the projects that contain the fields from the Project table.

1. Create a new file **ProjectsAPI.Page.al** and use snippet tpage (make sure to choose the proper one for API) and crate page **"MNB Projects API"**
2. Define mandatory properties for the API
3. Use **Job** table as a source and **System Id** as **ODataKeyFields**
4. Add proper columns from the Job table as fields. Remember to add captions to **System Id** and **No.** fields. Make sure that the **System Id** is not editable
5. Make sure that it is not possible to delete the projects
6. Add a column showing the customer's country code
7. Use Power Automate or any other tool to read and insert data into the projects



SOLUTION

```

page 50103 "MNB Projects API"
{
    PageType = API;
    Caption = 'projects';
    APIPublisher = 'mynavblog';
    APIGroup = 'custom';
    APIVersion = 'v1.0';
    EntityName = 'project';
    EntitySetName = 'projects';
    SourceTable = Job;
}

```



```

DelayedInsert = true;
ODataKeyFields = SystemId;
DeleteAllowed = false;

layout
{
    area(Content)
    {
        repeater(GroupName)
        {
            field(id; Rec.SystemId)
            {
                Caption = 'id';
                Editable = false;
            }
            field(number; Rec."No.")
            {
                Caption = 'number';
            }
            field(description; Rec.Description) { }
            field(description2; Rec."Description 2") { }
            field(startingDate; Rec."Starting Date") { }
            field(endingDate; Rec."Ending Date") { }
            field(billToCustomerNumber; Rec."Bill-to Customer No.")
            { }

            field(billToCustomerName; Rec."Bill-to Name") { }
            field(billToAddress; Rec."Bill-to Address") { }
            field(status; Rec.Status) { }
        }
    }
}

```

BOUND ACTIONS ON PAGE APIs

It is possible to add the actions to your custom APIs that can run by external sources. One of the examples can be posting a journal or document. Such actions are already defined in some of the standard APIs. Below you can find an example from Sales Orders API that allows to ShipAndInvoice use API.

```

[ServiceEnabled]
[Scope('Cloud')]
procedure ShipAndInvoice(var ActionContext: WebserviceActionContext)
var
    SalesHeader: Record "Sales Header";
    SalesInvoiceHeader: Record "Sales Invoice Header";
    SalesInvoiceAggregator: Codeunit "Sales Invoice Aggregator";
    Invoiced: Boolean;
begin
    GetOrder(SalesHeader);
    Invoiced := PostWithShipAndInvoice(SalesHeader, SalesInvoiceHeader);
    if Invoiced then
        SetActionResponse(ActionContext, SalesInvoiceAggregator.GetSalesInvoiceHeaderId(SalesInvoiceHeader), Page::"APIV2 - Sales Invoices", WebserviceActionResultCode::Deleted)
    else
        SetActionResponse(ActionContext, SalesHeader.SystemId, Page::"APIV2 - Sales Orders", WebserviceActionResultCode::Updated);
end;

```

To be able to use procedure as action it is required to set procedure as **ServiceEnabled**.

In return, the ActionContext is set to update if the document still exists or deleted if was posted.

```

local procedure SetActionResponse(var ActionContext: WebserviceActionContext; DocumentId: Guid; ObjectId: Integer; ResultCode: WebserviceActionResultCode)
begin
    ActionContext.SetObjectType(ObjectType::Page);
    ActionContext.SetObjectId(ObjectId);
    ActionContext.AddEntityKey(Rec.FieldNo(Id), DocumentId);
    ActionContext.SetResultCode(ResultCode);
end;

```



TASK: POSTING ITEM JOURNAL USING API

The customer asked you to create an API that will allow them to post the Item Journal Lines from a journal that is specified in the Inventory Setup.

1. Create a new file **InventorySetup.TabExt.al** that extends the **"Inventory Setup"** table and adds fields **MNB API Item Journal Template** and **MNB API Item Journal Batch**. Add proper table relations for easier input
2. Create a new file **InventorySetup.PagExt.al** and add the fields from the table extension to the new FastTab called **Integration**
3. Create a new API for the **Item Journal Line**. You may add new fields for the API but for the test, it will not be necessary to insert the lines to the journal only mandatory field is **SystemId**
4. Add ODataKeyFields so only the System Id field is needed
5. Create a new procedure to Post Journal that is set as **ServiceEnabled** procedure
6. Test if the setup fields have value and post the journal for the proper Batch

7. Make sure to set an action context to deleted when lines have been posted and none in other case



SOLUTION

```
page 50105 "MNB Item Journal Line API"
{
    PageType = API;
    Caption = 'itemJorunal';
    APIPublisher = 'mynavblog';
    APIGroup = 'custom';
    APIVersion = 'v1.0';
    EntityName = 'itemJournalLine';
    EntitySetName = 'itemJournalLines';
    SourceTable = "Item Journal Line";
    DelayedInsert = true;
    ODataKeyFields = SystemId;

    layout
    {
        area(Content)
        {
            repeater(GroupName)
            {
                field(id; Rec.SystemId)
                {
                    Caption = 'id';
                    Editable = false;
                }
                field(type; Rec.Type) { }
                field(documentNo; Rec."Document No.") { }
                field(lineNo; Rec."Line No.") { }
                field(documentDate; Rec."Document Date") { }
                field(postingDate; Rec."Posting Date") { }
                field(number; Rec."No.") { }
                field(quantity; Rec.Quantity) { }
                field(unitOfMeasureCode; Rec."Unit of Measure Code") { }
            }
        }
    }
    trigger OnOpenPage()
    var
        InventorySetup: Record "Inventory Setup";
    begin
        InventorySetup.Get();
        InventorySetup.TestField("MNB API Item Journal Template");
    end
}
```

```

        InventorySetup.TestField("MNB API Item Journal Batch");
        Rec.SetRange("Journal Template Name", InventorySetup."MNB API
Item Journal Template");
        Rec.SetRange("Journal Batch Name", InventorySetup."MNB API Item
Journal Batch");
    end;

    local procedure SetActionResponse(var ActionContext:
WebServiceActionContext; PageId: Integer; RecId: Guid; ResultCode:
WebServiceActionResultCode)
    var
    begin
        ActionContext.SetObjectType(ObjectType::Page);
        ActionContext.SetObjectId(PageId);
        ActionContext.AddEntityKey(Rec.FieldNo(SystemId), RecId);
        ActionContext.SetResultCode(ResultCode);
    end;

    [ServiceEnabled]
    procedure PostJournal(var ActionContext: WebServiceActionContext)
    var
        InventorySetup: Record "Inventory Setup";
        ItemJournalLine: Record "Item Journal Line";
        ItemJnlPostBatch: Codeunit "Item Jnl.-Post Batch";
    begin
        InventorySetup.Get();
        InventorySetup.TestField("MNB API Item Journal Template");
        InventorySetup.TestField("MNB API Item Journal Batch");
        ItemJournalLine.SetRange("Journal Template Name",
InventorySetup."MNB API Item Journal Template");
        ItemJournalLine.SetRange("Journal Batch Name",
InventorySetup."MNB API Item Journal Batch");
        if ItemJournalLine.FindFirst() then
            if ItemJnlPostBatch.Run(ItemJournalLine) then
                SetActionResponse(ActionContext, Page::"MNB Item Journal
Line API", ItemJournalLine.SystemId,
WebServiceActionResultCode::Deleted)
            else
                SetActionResponse(ActionContext, Page::"MNB Item Journal
Line API", ItemJournalLine.SystemId, WebServiceActionResultCode::None);
            end;
    }

```

QUERY APIS

APIs based on query are used for reading data from Business Central, for example for Power BI reports without any additional development.



TASK: CUSTOMER SALES API

The customer was happy with the view that was added previously to show the amount and remaining for the invoices from the Countries/Region page. Now the company would like also similar data to be able to be viewed from Power BI. Based on the previous query ("**MNB Customer Ledg. by Country**") new one needs to be created that shows also the customer name and detailed address.

1. Create a new file **CustomerSalesAPI.Que.al** and create API query using **tquery** snippet.

Remember to use proper snippet with API type

2. Add columns the same as in "**MNB Customer Ledg. by Country**" but remember about camel cases.
3. Add new columns for Customer No., Name, Address, Post Code
4. Try to run the query from Power BI

The screenshot shows the 'Nawigator' window in Microsoft Dynamics NAV. On the left, under 'Opcje wyświetlania', a list of queries is shown. 'customersSales' is selected and highlighted. On the right, the 'customersSales' query results are displayed in a table. The table has five columns: 'postingDateMonth', 'postingDateYear', 'customerNo', 'amount', and 'remaining'. The data is organized by year (2023 and 2024) and then by month (1, 2, 3). Each row shows the customer number, the amount, and the remaining balance.

postingDateMonth	postingDateYear	customerNo	amount	remaining
1	2023	10000	12449,27	
1	2023	20000	3403,48	
1	2023	30000	12053,66	
1	2023	40000	4526,1	
1	2023	50000	5351,52	
1	2024	10000	19801,44	
1	2024	20000	10215,46	
1	2024	30000	16074,08	
1	2024	40000	10233,16	
1	2024	50000	6038,19	
2	2023	10000	14570,97	
2	2023	20000	3808,36	
2	2023	30000	15411,97	
2	2023	40000	3832,95	
2	2023	50000	6270,75	
2	2024	10000	21908,92	
2	2024	20000	7874,38	
2	2024	30000	20822,52	
2	2024	40000	4729,51	
2	2024	50000	7876,65	
3	2023	10000	18999,97	
3	2023	20000	4612,45	
3	2023	30000	17759,86	



SOLUTION

```

query 50102 "MNB Customer Sales API"
{
    QueryType = API;
    APIPublisher = 'mynavblog';
    APIGroup = 'custom';
    APIVersion = 'v1.0';
    EntityName = 'customerSales';
    EntitySetName = 'customersSales';
    DataAccessIntent = ReadOnly;

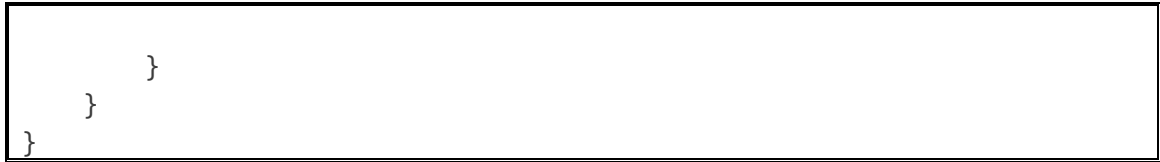
    elements
    {
        dataitem(Cust__Ledger_Entry; "Cust. Ledger Entry")
        {
            DataItemTableFilter = "Document Type" = const(Invoice);

            column(postingDateMonth; "Posting Date")
            {
                Method = Month;
            }
            column(postingDateYear; "Posting Date")
            {
                Method = Year;
            }
            column(amount; Amount)
            {
                Method = Sum;
            }
            column(remainingAmount; "Remaining Amount")
            {
                Method = Sum;
            }
            column(customerNo; "Customer No.") { }

            dataitem(Customer; "Customer")
            {
                DataItemLink = "No." = Cust__Ledger_Entry."Customer
No.";

                column(name; Name) { }
                column(countryRegionCode; "Country/Region Code") { }
                column(county; County) { }
                column(city; City) { }
                column(address; Address) { }
                column(postCode; "Post Code") { }
            }
        }
    }
}

```



EXTERNAL BUSINESS EVENTS

External Business Events are very useful when any integration is needed when some action happens inside Business Central. Some examples of such actions you may find below:

- ✓ Change the Status of the document to Released
- ✓ Post Sales or Purchase Order
- ✓ Block the Customer or Vendor

In the above examples, the process may require some external actions like:

- ✓ Sending Email to the vendor or customer
- ✓ Notify the warehouse employees
- ✓ Change external systems

In such cases, External Business Events can be useful to trigger the flow that should happen. It is possible to use them inside Power Automate or any other external software that can listen to the changes from Business Central.

Creating External Business Events is very similar to creating events inside AL – the procedure needs to be specified as [ExternalBusinessEvent]. To define an event you need to specify:

- ✓ Name
- ✓ Name that is displayed
- ✓ Description
- ✓ Event Category – it requires either adding dependency to extension with standard Business Events or extending the **EventCategory** enum

The External Business Event can contain the parameters visible when using the event. The good practice is to add the **System Id** of the record from which the event has been triggered. For example

when adding an event when the customer is blocked then the **System Id** of the customer will help to retrieve the record for the next steps of the flow.



TASK: CUSTOMER READY FOR SYNCHRONIZATION

The user wants to synchronize the records of the Customer with an external application. However, they want to do it only when the record is ready for synchronization. They would like to mark that the record is ready and then synchronization should take place. Use External Business Event for it.

1. Create a new enum extension for the **EventCategory** enum. Add new value "MNB Custom"
2. Create a new codeunit "**MNB Sync Customer Event**" and add a new External Business Event **CustomerReadyForSync**. As parameters add **customerId**, **customer**, and **customer name**
3. Open the **Customer** table extension file and add a new field "**MNB Ready For Sync.**" type Boolean. Add that when the value of the field is true then the external business event is triggered
4. Add the new field to the Customer Card page
5. Try to create a flow in Power Automate

The screenshot displays a Power Automate flow configuration with two steps connected by a downward arrow.

Step 1: When a business event occurs (V3) (Preview)

- Environment: KBISANDBOX231
- Event: Customer Ready For Sync. (mynavblog.com)
- Company (optional): CRONUS USA, Inc.

Step 2: Get record (V3)

- Environment: KBISANDBOX231
- Company: CRONUS USA, Inc.
- API category: v2.0
- Table name: customers
- Row id: customerId x

At the bottom, there are buttons for "+ New step" and "Save".



SOLUTION

```
enumextension 50100 "MNB Event Category" extends EventCategory
{
    value(50100; "MNB Custom")
    {
        Caption = 'Custom';
    }
}
```

```
codeunit 50115 "MNB Sync Customer Event"
{
    [ExternalBusinessEvent('MNBCustomerReadyForSync', 'Customer Ready
    For Sync.', 'Triggered when a customer is set that can be
    synchronized.', EventCategory::"MNB Custom")]
    procedure CustomerReadyForSync(customerId: Guid; customerNo:
    Code[20]; customerName: Text[100])
    begin
    end;
}
```

```
field(50101; "MNB Ready For Sync."; Boolean)
{
    Caption = 'Ready For Sync.';
    DataClassification = SystemMetadata;
    trigger OnValidate()
    var
        SyncCustomerEvent: Codeunit "MNB Sync Customer Event";
    begin
        if Rec."MNB Ready For Sync." then
            SyncCustomerEvent.CustomerReadyForSync(Rec.SystemId,
            Rec."No.", Rec.Name);
        end;
    }
}
```

```
layout
{
    addlast(General)
    {
        field("MNB Ready For Sync."; Rec."MNB Ready For Sync.")
        {
            ApplicationArea = All;
        }
    }
}
```

```
        Tooltip = 'Set if the customer is ready for sync with  
external software.';  
    }  
}  
}
```

CHAPTER SUMMARY

- ✓ In this chapter, you learned how to create APIs for Pages and Queries
- ✓ You developed API allowing you to insert data to the system, run actions, and read data for further integration
- ✓ You also created an External Business Event that is triggered whenever action occurs

CHAPTER 7

BACKGROUND TASKS

OBJECTIVES

In some cases, the async operation needs to be done in the Business Central process. In this chapter, the main goals are to

- ✓ Learn how to run tasks in the background
- ✓ Learn how to create a page background tasks

PAGE BACKGROUND TASKS

In some cases, it is need to calculate some of the data when opening the page. That operation may take a while when there is some complicated calculation involved. Each such calculation may stop a customer from working in the system. Therefore from a performance perspective, it may be needed, instead of calculating the value during a parent session, to run it as a background task in a child session.

The page background task can be triggered from the page and update the value of the variable in the background. The important fact is that the page background task can only do read-only operations, which means it cannot write transactions or lock the database.



EXAMPLE – ENQUEUE PAGE BACKGROUND TASK

The below example shows how to enqueue the page background task. The **EnqueueBackgroundTask** method defines which codeunit should be run. It is possible to send to the codeunit the task parameters using a dictionary as below.

```
trigger OnAfterGetCurrRecord()
var
    MNBICalculateRisk: Interface MNBICalculateRisk;
    TaskParameters: Dictionary of [Text, Text];
begin
    TaskParameters.Add('CustomerNo', Rec."No.");
    CurrPage.EnqueueBackgroundTask(BackgroundTaskId, Codeunit::"MNB
Check Customer In Register", TaskParameters)
end;
```

When the background task is enqueued the child session is open and executes the code in the OnRun trigger of the specified codeunit. It is possible to get the value of the parameters sent to the codeunit using **Page.GetBackgroundParameters()**.

After the calculation, the task results can be set using the dictionary and the result needs to be sent to the page using the **SetBackgroundTaskResult** method.

```
codeunit "MNB Check Customer In Register"
{
    trigger OnRun()
    var
        TaskResults: Dictionary of [Text, Text];
        CustomerNo: Code[20];
    begin
```

```

        CustomerNo := Page.GetBackgroundParameters().Get('CustomerNo');
        ...
        TaskResults.Add('CustomerIsOk', format(true));
        Page.SetBackgroundTaskResult(TaskResults);
    end;
}

```

The result can be displayed on the page using the trigger **OnPageBackgroundTaskCompleted** which contains the results of the page background task. In the example below the global variable **CustomerIsOk** is set in the trigger.

If there are any errors you can display notifications using the **OnPageBackgroundTaskError** trigger.

```

trigger OnPageBackgroundTaskCompleted(TaskId: Integer; Results:
Dictionary of [Text, Text])
begin
    Evaluate(CustomerIsOk, Results.Get('CustomerIsOk'));
end;

trigger OnPageBackgroundTaskError(TaskId: Integer; ErrorCode: Text;
ErrorText: Text; ErrorCallStack: Text; var IsHandled: Boolean)
begin
    //Display notifications about the error
end;

```



TASK: GET ITEM EXTERNAL INVENTORY

The customer has multiple companies and would like to see the total stock for the item in all companies when opening the Item Card.

1. Create the new codeunit "**MNB Check Item Availability**" and add Dictionary of [Text, Text] to the OnRun trigger. You may call it **TaskResults**
2. Create a new variable **ItemNo** – type Code[20] and assign the value using **Page.GetBackgroundParameters()** – get **ItemNo** key
3. Create a new local procedure **GetExernalInventory** which as a parameter has an Item Number. Return Text from the function
4. In the function, you may exit with a random number to show the value on the page

5. In the OnRun trigger run the local procedure **GetExernalInventory** and add to the **TaskResults ExternalInventory** key and value that is calculated in the function
6. Open the Item Card extension and add a new global decimal variable **ExternalInventory**.
Show it as last on Item FastTab. Make it not editable.
7. Create a Global integer variable called **BackgroundTaskId**
8. In the **OnAfterGetCurrRecord** trigger add **TaskParameters** Dictionary of [Text, Text] and add to it **ItemNo** key with the "No." field as the value
9. In the same trigger enqueue the background task. Remember to specify proper codeunit and task parameters
10. Add a new trigger OnPageBackgroundTaskCompleted and evaluate the value of the result **ExternalInventory** dictionary key to the global variable **ExternalInventory**



SOLUTION

```
codeunit 50124 "MNB Check Item Availability"
{
    trigger OnRun()
    var
        TaskResults: Dictionary of [Text, Text];
    begin
        TaskResults.Add('ExternalInventory',
            GetExternalInventory(Page.GetBackgroundParameters().Get('ItemNo')));
        Page.SetBackgroundTaskResult(TaskResults);
    end;

    local procedure GetExternalInventory(ItemNo: Code[20]): Text
    begin
        exit(Format(1005));
    end;
}
```

```
addlast(Item)
{
    field(MNBExternalInventoryField; ExternalInventory)
    {
        ApplicationArea = All;
    }
}
```

```

        Caption = 'External Inventory';
        ToolTip = 'Displays the external inventory for the item.';
        Editable = false;
    }
}
...
var
    BackgroundTaskId: Integer;
    ExternalInventory: Decimal;

...
trigger OnAfterGetCurrRecord()
var
    MNBICalculateRisk: Interface MNBICalculateRisk;
    TaskParameters: Dictionary of [Text, Text];
begin
    TaskParameters.Add('ItemNo', Rec."No.");
    CurrPage.EnqueueBackgroundTask(BackgroundTaskId, Codeunit::"MNB
Check Item Availability", TaskParameters)
end;

trigger OnPageBackgroundTaskCompleted(TaskId: Integer; Results:
Dictionary of [Text, Text])
begin
    Evaluate(ExternalInventory, Results.Get('ExternalInventory'));
end;

```

CHAPTER SUMMARY

- ✓ In this chapter, how to create a page background tasks
- ✓ You developed functionality that gets external inventory in the background

CHAPTER 8

HANDLING CSV, XML, AND JSON FILES FROM THE CODE

OBJECTIVES

In many cases, it is necessary to export or import all sorts of files when working with Business Central.

The main goal of this chapter is to learn how to

- ✓ Import and Export different types of files using AL
- ✓ Learn how to work with InStreams and OutStreams

INTRODUCTION

For working with CSV, TXT or XML files the XMLPort object can be used. However, in this workbook, other tools that can be found in AL will be used to show alternative approaches that in some cases can be easier to implement and more efficient.

INSTREAM, OUTSTREAM, AND TEMPBLOB CODEUNIT

When working with importing or exporting the files in AL it is necessary to work with streams. In AL it is possible to define two data types that are helpful for that

- ✓ **InStream** – is used to read the data to stream for example from imported files. Therefore you will find methods **Read**, and **ReadText** that allow you to read the value inside the stream (from the imported file).

- ✓ **OutStream** – is used to write the data to Stream and export the data for example to a file. Therefore you will find methods as **Write**, **WriteText** that allow you to write data into a stream.

If you need to upload a file to the stream (**InStream**) you can use the method **UploadIntoStream**. It allows importing files from the user's local computer directly to the stream.

A similar method, **DownloadFromStream**, can be used to retrieve data in a file from the stream. In this case, you also need to specify the **InStream** variable.

When working with importing the data from files, for example, XML, CSV, txt, JSON, etc.), into Business Central there need to be done operations in the below order:

- ✓ **Upload From Stream** – that uploads data into the InStream variable

- ✓ **Read InStream Data** - that reads the data from the stream

When working with exporting the data from Business Central to the file (for example XML, CSV, txt, JSON, etc.), in many cases there is a need to do operations in the below order:

- ✓ **Create OutStream** – that creates the stream to which data can be written

- ✓ **Write Data to OutStream** – that populates the stream with data
- ✓ **Create InStream from OutStream** – that creates InStream that later can be exported
- ✓ **Assing File Name** – that creates a name of the file that needs to be exported
- ✓ **Download From Steam** – that downloads the file to the local machine

As you can see there are more steps needed to export data using streams. However, there is **codeunit TempBlob** that is responsible for Creating **OutStream** and copying it to **InStream**.



EXAMPLE TEMPBLOB – EXPORT FILE

An example of the usage of **TempBlob** codeunit can be seen below. First, the OutStream is created and the text is written to it. Next, the InStream is created (in background the function copy stream copies OutStream to Instream). Later the file with the given name is exported from Business Central

```
procedure ExportFileWithStream()
var
    TempBlob: Codeunit "Temp Blob";
    OStream: OutStream;
    InStream: InStream;
    Filename: Text;
    FileNameTxt: Label 'testfile.txt', Locked = true;
begin
    TempBlob.CreateOutStream(OStream);
    OStream.WriteText('Text for Stream');
    TempBlob.CreateInStream(InStream);
    FileName := FileNameTxt;
    DownloadFromStream(InStream, '', '', '', FileName);
end;
```

CSV FILES

When working with CSV Files CSVBuffer temporary table can be used. The table contains procedures that allow an easy way to load and save data to stream (LoadDataFromStream and SaveDataToBlob).



TASK: EXPORT PAYMENT JOURNAL TO CSV

The customer would like to get an action that exports the Payment Journal lines to a CSV file. The file with a header should contain the below columns and should have a comma (,) as a separator:

- a. **Document Number** – from field Document No.
 - b. **Payment Amount** – from field Amount
 - c. **Currency Code** – from the field Currency Code
 - d. **Vendor Number** – from field Account No.
 - e. **Payment Date** – from field Posting Date in Format '<Day>/<Month,2>/<Year4>'
1. Create a new codeunit “**MNB Export Payment to CSV**” and add the procedure **ExportToCSV**.
The procedure should have a parameter for Record **Gen. Journal Line**
 2. Filter the general journal line and if lines are found create a header line using the CSVBuffer table function **Insert Entry**
 3. Add in the loop for all lines the line with data using CSVBuffer table function **Insert Entry**
 4. After the loop assign the FileName for the exported file as **PaymentJournal.csv**
 5. Use SaveDataToBlob from the CSVBuffer table to save the rows. Add comma as separator
 6. Use the method DownloadFromStream to download the value from InStream
 7. Add a new page extension for Payment Journal and add the action that will run the procedure from the created codeunit

```

1 | Document Number,Payment Amount,Currency Code,Vendor Number,Payment Date
2 | G04001,12312.31,,40000,8/04/2024
3 | G04002,1232.00,CAD,10000,8/04/2024
4

```



SOLUTION

```

codeunit 50116 "MNB Export Payment to CSV"
{
    internal procedure ExportToCSV(var GenJournalLine: Record "Gen.
Journal Line")
    var
        GenJnlLine: Record "Gen. Journal Line";
        CSVBuffer: Record "CSV Buffer" temporary;
        TempBlob: Codeunit "Temp Blob";
        FileName: Text;

```

```

        LineNo: Integer;
        FileNameLbl: Label 'PaymentJournal.csv', Locked = true;
        ThereAreNoLineErr: Label 'There are no lines in the journal.';
    begin
        GenJnlLine.SetRange("Journal Template Name",
GenJournalLine."Journal Template Name");
        GenJnlLine.SetRange("Journal Batch Name",
GenJournalLine."Journal Batch Name");

        if not GenJnlLine.FindSet() then
            Error(ThereAreNoLineErr);

        LineNo := 1;

        InsertHeader(CSVBuffer, LineNo);

        repeat
            InsertLine(GenJnlLine, CSVBuffer, LineNo);
        until GenJnlLine.Next() = 0;

        FileName := FileNameLbl;
        CSVBuffer.SaveDataToBlob(TempBlob, ',');
        DownloadFromStream(TempBlob.CreateInStream(), '', '', '',
FileName);
    end;

    local procedure InsertHeader(var CSVBuffer: Record "CSV Buffer"
temporary; var LineNo: Integer)
    var
        DocumentNumberLbl: Label 'Document Number';
        PaymentAmountLbl: Label 'Payment Amount';
        CurrencyCodeLbl: Label 'Currency Code';
        VendorNumberLbl: Label 'Vendor Number';
        PaymentDateLbl: Label 'Payment Date';
    begin
        CSVBuffer.InsertEntry(LineNo, 1, DocumentNumberLbl);
        CSVBuffer.InsertEntry(LineNo, 2, PaymentAmountLbl);
        CSVBuffer.InsertEntry(LineNo, 3, CurrencyCodeLbl);
        CSVBuffer.InsertEntry(LineNo, 4, VendorNumberLbl);
        CSVBuffer.InsertEntry(LineNo, 5, PaymentDateLbl);
        LineNo += 1;
    end;

    local procedure InsertLine(GenJournalLine: Record "Gen. Journal
Line"; var CSVBuffer: Record "CSV Buffer" temporary; var LineNo:
Integer)
    begin
        CSVBuffer.InsertEntry(LineNo, 1, GenJournalLine."Document No.");

```

```

        CSVBuffer.InsertEntry(LineNo, 2, Format(GenJournalLine.Amount,
0, 9));
        CSVBuffer.InsertEntry(LineNo, 3, GenJournalLine."Currency
Code");
        CSVBuffer.InsertEntry(LineNo, 4, GenJournalLine."Account No.");
        CSVBuffer.InsertEntry(LineNo, 5, Format(GenJournalLine."Posting
Date", 0, '<Day>/<Month,2>/<Year4>'));
        LineNo += 1;
    end;
}

```

```

pageextension 50109 "MNB Payment Journal" extends "Payment Journal"
{
    actions
    {
        addlast(Processing)
        {
            action(MNBExportToCSV)
            {
                ApplicationArea = All;
                Caption = 'Export to CSV';
                Image = ExportFile;
                Tooltip = 'Exports the payment journal to a CSV file for
finance reporting.';
                trigger OnAction()
                var
                    ExportPaymentToCSV: Codeunit "MNB Export Payment to
CSV";

                begin
                    ExportPaymentToCSV.ExportToCSV(Rec);
                end;
            }
        }
        addfirst(Category_Process)
        {
            actionref(MNBExportToCSV_Promoted; MNBExportToCSV)
            {
            }
        }
    }
}

```

JSON FILES

When writing and reading Json files in AL it is possible to use specific Json object types such as **JSONArray**, **JsonObject**, **JsonToken**, and **JsonValue**. In this workbook, however, a simplified method will be used – using **JSON Text Reader/Writer** codeunit that has all useful methods allowing to creation of JSON files. For reading the files the temporary table **Json Buffer** will be used.



TASK: EXPORT CUSTOMERS TO JSON FILE

The user would like to get an action that exports customers in the below structure. Use **JSON Text Reader/Writer** for easier creation of the file.

```
1  {
2      "customers": [
3          {
4              "name": "Adatum Corporation",
5              "number": "10000",
6              "blocked": false,
7              "balance": 0.0,
8              "creditLimit": 0.0
9          },
10         {
11             "name": "Trey Research",
12             "number": "20000",
13             "blocked": false,
14             "balance": 3036.6,
15             "creditLimit": 0.0
16         }
17     ]
18 }
```

1. Create a new codeunit “**MNB Export Customers Json**” and add procedure **ExportFromJson**.
2. Using **Json Text Reader/Writer** codeunit start an empty object and create a new array of ‘customers’
3. In the loop for customers start new empty objects and add properties:

- a. name – from Customer Name
 - b. number – from Customer No.
 - c. blocked – from Customer Privacy Blocked
 - d. balance – from Customer Balance
 - e. creditLimit - from Customer Credit Limit
4. End object in the loop
5. Outside the loop end array and end object
6. Using TempBlob Codeunit create OutStream and write text value from **Json Text Reader/Writer** to it
7. Create file name customers.json and download InStream
8. Create a new page extension for the Customer List and add a new action to export the Customers. Run the procedure from the created codeunit

SOLUTION

```
codeunit 50117 "MNB Export Customers Json"
{
    internal procedure GenerateJsonForCustomers()
    var
        Customer: Record Customer;
        JsonTextReaderWriter: Codeunit "Json Text Reader/Writer";
        TempBlob: Codeunit "Temp Blob";
        OStream: OutStream;
        FileName: Text;
        FileNameTxt: Label 'customers.json';
    begin
        JsonTextReaderWriter.WriteStartObject('');
        JsonTextReaderWriter.WriteStartArray('customers');
        Customer.SetAutoCalcFields(Balance);
        if Customer.FindSet() then
            repeat
                JsonTextReaderWriter.WriteStartObject('');
                JsonTextReaderWriter.WriteStringProperty('name',
Customer.Name);
                JsonTextReaderWriter.WriteStringProperty('number',
Customer."No.");
```

```

        JsonTextWriterWriter.WriteBooleanProperty('blocked',
Customer."Privacy Blocked");
        JsonTextWriterWriter.WriteNumberProperty('balance',
Customer.Balance);
        JsonTextWriterWriter.WriteNumberProperty('creditLimit',
Customer."Credit Limit (LCY)");
        JsonTextWriterWriter.WriteEndObject();
        until Customer.Next() = 0;
        JsonTextWriterWriter.WriteEndArray();
        JsonTextWriterWriter.WriteEndObject();

        TempBlob.CreateOutputStream(OSTeam);
        OSTeam.WriteText(JsonTextWriterWriter.GetJsonAsText());
        FileName := FileNameTxt;
        DownloadFromStream(TempBlob.CreateInStream(), '', '', '',
FileName);
        end;
    }

```

```

pageextension 50110 "MNB Customer List" extends "Customer List"
{
    actions
    {
        addlast(processing)
        {
            action(MNBExportToJson)
            {
                ApplicationArea = All;
                Caption = 'Export to JSON';
                Image = ExportFile;
                ToolTip = 'Exports the customer list to a JSON file.';

                trigger OnAction()
                var
                    ExportCustomersJson: Codeunit "MNB Export Customers
Json";

                begin
                    ExportCustomersJson.GenerateJsonForCustomers();
                end;
            }
        }
        addlast(Category_Process)
        {
            actionref(MNBExportToJson_Promoted; MNBExportToJson)
            {
            }
        }
    }
}

```



```
}  
}  
}
```



TASK: IMPORT ITEMS FROM THE JSON FILE

The user needs the action that imports items directly to Business Central. Use the **Json Buffer** table for easier reading the file. The structure of the file is below – you can use it as an example to import after saving it to a local machine as a JSON file.

```
{  
  "items": [  
    {  
      "name": "Item 1",  
      "number": "A-001",  
      "baseUnitOfMeasure": "PCS",  
      "variants": []  
    },  
    {  
      "name": "Item 2",  
      "number": "A-002",  
      "baseUnitOfMeasure": "PCS",  
      "variants": [  
        {  
          "code": "S-1",  
          "description": "Small"  
        },  
        {  
          "code": "M-1",  
          "description": "Medium"  
        }  
      ]  
    },  
    {  
      "name": "Item 3",  
      "number": "A-003",  
      "baseUnitOfMeasure": "PCS",  
      "variants": []  
    }  
  ]  
}
```

HINT

When importing the files to Business Central good practice is to create a buffer table to not create directly the records. However, for this task, this will be skipped since this is an example of importing JSON files. If you have time and would like to make this real World scenario please create a buffer table and then insert the records into the Item table after validation.

1. Create a new codeunit “**MNB Import Items Json**” and add procedure **ImportFromJson**.
2. Use the **UploadIntoStream** method to get the value of the JSON file to **InStream**
3. Read **InStream** text using loop and add the value to the TextBuilder variable
4. Use the function **ReadFromText** from the **JSON Buffer** table to read the TextBuilder variable and populate the table
5. Use **JSON Buffer** table to find array **items** and read properties in the loop for **number**, **description**, **baseUnitOfMeasure** to variables and create a new item
6. Inside the loop for Items try to find array variants, if exists in loop find variables for **code** and **description**, and create variants
7. Open the page extension for Item List and add the action to import the JSON file

SOLUTION

```
codeunit 50118 "MNB Import Items Json"
{
    procedure ImportItemsFromJson()
    var
        JSONBuffer, JSONBufferItems, JSONBufferVariants: Record "JSON
Buffer" temporary;
        IStream: InStream;
        FileName: Text;
        ItemNo, ItemDescription, BaseUnitOfMeasure, VariantCode,
VariantDescription: Text;
        JsonTextBuilder: TextBuilder;
    begin
        UploadIntoStream('', '', '', FileName, IStream);
```

```

        GetStreamText(IStream, JsonTextBuilder);
        JSONBuffer.ReadFromText(JsonTextBuilder.ToText());
        if JSONBuffer.FindArray(JSONBufferItems, 'items') then
            repeat
                JSONBufferItems.GetPropertyValue(ItemNo, 'number');
                JSONBufferItems.GetPropertyValue(ItemDescription,
'name');
                JSONBufferItems.GetPropertyValue(BaseUnitOfMeasure,
'baseUnitOfMeasure');
                CreateItem(ItemNo, ItemDescription, BaseUnitOfMeasure);
                if JSONBufferItems.FindArray(JSONBufferVariants,
'variants') then
                    repeat
                        JSONBufferVariants.GetPropertyValue(VariantCode,
'code');
                        JSONBufferVariants.GetPropertyValue(VariantDescr
iption, 'description');
                        CreateVariant(ItemNo, VariantCode,
VariantDescription);
                    until JSONBufferVariants.Next() = 0;
                until JSONBufferItems.Next() = 0;
            end;

        local procedure GetStreamText(var IStream: InStream; var JsonText:
TextBuilder)
        var
            TempText: Text;
        begin
            while not (IStream.EOS) do begin
                IStream.ReadText(TempText);
                JsonText.Append(TempText);
            end;
        end;

        local procedure CreateItem(ItemNo: Text; ItemDescription: Text;
BaseUnitOfMeasure: Text)
        var
            Item: Record Item;
        begin
            Item.Init();
            Item.Validate("No.", ItemNo);
            Item.Validate("Description", ItemDescription);
            Item.Insert(true);
            Item.Validate("Base Unit of Measure", BaseUnitOfMeasure);
            Item.Modify(true);
        end;

```

```

    local procedure CreateVariant(ItemNo: Text; VariantCode: Text;
VariantDescription: Text)
    var
        ItemVariant: Record "Item Variant";
    begin
        ItemVariant.Init();
        ItemVariant.Validate("Item No.", ItemNo);
        ItemVariant.Validate("Code", VariantCode);
        ItemVariant.Validate("Description", VariantDescription);
        ItemVariant.Insert(true);
    end;
}

```

```

pageextension 50103 "MNB Item List" extends "Item List"
{
    actions
    {
        addlast(Functions)
        {
            action(MNBImportJson)
            {
                ApplicationArea = All;
                Caption = 'Import From Json';
                Image = Import;
                Tooltip = 'Import items from Json file.';
                trigger OnAction()
                var
                    ImportItemsJson: Codeunit "MNB Import Items Json";
                begin
                    ImportItemsJson.ImportItemsFromJson();
                end;
            }
        }
        addlast(Category_Process)
        {
            actionref(MNBImportJson_Promoted; MNBImportJson)
            {
            }
        }
    }
}

```

CHAPTER SUMMARY

- ✓ In this chapter, you learn about the List and Dictionary data types.
- ✓ You developed functionalities that required to use of those types.

CHAPTER 9

USEFUL MODULES FROM SYSTEMAPP

OBJECTIVES

In this chapter, you will learn where to find standard modules that as a developer you can use during your development tasks. You will:

- ✓ Learn what is SystemApp
- ✓ Develop integration with Azure Functions
- ✓ Develop integration with Azure Blob Storage

INTRODUCTION

Business Central has many useful modules that can be used out of the box by AL developers. Each module focuses on different aspects of system operations and user interactions, providing functionalities for data handling, communication, task automation, and report generation within the application.

All system modules can be found on GitHub:

<https://github.com/microsoft/BCApps/tree/main/src/System%20Application/App>

Modules can be added by Microsoft, but also by partners and freelancers. Everyone is allowed to help build new functionalities.

Among existing modules, you can find ones such as:

- ✓ Integration with SharePoint
- ✓ Barcodes
- ✓ Integration with Azure Functions
- ✓ Integration with Azure Blob Storage
- ✓ Base64
- ✓ Rest Client
- ✓ Camera and Image Interaction




And many more. In this workbook, only a few will be described

STRUCTURE OF THE MODULES

Based on the simple module of Base64 Converter you may find how each module is structured. You can find the module here:

<https://github.com/microsoft/BCApps/tree/main/src/System%20Application/App/Base64%20Converter/src>

In the module, you will find the **src** folder that contains the functionality.

Name
 ..
 Base64Convert.Codeunit.al
 Base64ConvertImpl.Codeunit.al

The SystemApp modules are built using a facade pattern and expose only the functions that developers can use. You can find those functions in the first codeunit. The codeunit with suffix **Impl.** contains the code that is accessed only internally (by Microsoft) and contains the logic.

In other words, you can use only procedures from codeunit that is not implementation codeunit.

```

1  // -----
2  // Copyright (c) Microsoft Corporation. All rights reserved.
3  // Licensed under the MIT License. See License.txt in the project root for license information.
4  // -----
5
6  namespace System.Text;
7
8  /// <summary>
9  /// Converts text to and from its base-64 representation.
10 /// </summary>
11 codeunit 4110 "Base64 Convert"
12 {
13     Access = Public;
14     SingleInstance = true;
15     InherentEntitlements = X;
16     InherentPermissions = X;
17
18     var
19         Base64ConvertImpl: Codeunit "Base64 Convert Impl.";
20
21     /// <summary>
22     /// Converts the value of the input string to its equivalent string representation that is encoded with base-64 digits.
23     /// </summary>
24     /// <param name="String">The string to convert.</param>
25     /// <returns>The string representation, in base-64, of the input string.</returns>
26     procedure ToBase64(String: Text): Text
27     begin
28         exit(Base64ConvertImpl.ToBase64(String));
29     end;

```

AZURE FUNCTION

In brief Azure Functions is a serverless computing service that allows you to run event-driven code without having to manage infrastructure. They can be used to build HTTP-based API endpoints,

process data streams in real time, integrate systems with automated workflows, and run scheduled tasks. Azure Functions support various programming languages and can be triggered by different events such as HTTP requests, timer schedules, and messages from Azure services like Storage Queues, Service Bus, and Event Hubs, making them highly versatile for cloud-based automation and microservices architecture.

In this workbook, tasks will not require any knowledge about Azure Functions but will allow to consumption of the data retrieved from Azure Functions.

In the repository for this workbook, you can find code for the Azure Function that is used for the task



TASK: GET ITEM EXTERNAL INVENTORY (FROM AZURE FUNCTION)

This task is an extension to the task from the Background Tasks chapter. Now instead of a hardcoded value, you will connect to azure function that returns the quantity in the external system.

1. Open the codeunit "**MNB Check Item Availability**"
2. Create a new local procedure **GetExernalInventoryFromAzureFunction** which as a parameter has an Item Number. Return Text from the function
3. In the function use the "Azure Functions Authentication" codeunit to create authentication to the Azure function. As an endpoint, you may use below address (it does not require any authentication)

<https://itemimporttobc.azurewebsites.net/api/GetItemsInventoryToBC?code=3xRDnOyU8rREMIK9j8CEq35xPsR9OxMGmG9t4ljef84AzFuHzOBw==>
4. Create a new variable type Dictionary and add two keys: **itemnumber** (with value the same as Item Number), **secretkey** (with value 'altraining'). Please make sure that the values and keys have the same cases as above
5. Use Azure Functions Response codeunit to get a response as text
6. Use the function in **OnRun** codeunit to retrieve the value and send it with background task as previously



SOLUTION

```
procedure GetExternalInventoryAzureFunction(ItemNo: Code[20]): Text
var
    AzureFunctionsAuthentication: Codeunit "Azure Functions
Authentication";
    AzureFunctions: Codeunit "Azure Functions";
    AzureFunctionsResponse: Codeunit "Azure Functions Response";
    AzureFunctionsAuthenticationIn: Interface "Azure Functions
Authentication";
    QueryDict: Dictionary of [Text, Text];
    ResponseTxt: Text;
    AzureFunctionUrlTxt: Label 'Create ', Locked = true;
begin
    AzureFunctionsAuthenticationIn :=
AzureFunctionsAuthentication.CreateCodeAuth(AzureFunctionUrlTxt, '');
    QueryDict.Add('itemnumber', ItemNo);
    QueryDict.Add('secretkey', 'altraining');
    AzureFunctionsResponse :=
AzureFunctions.SendGetRequest(AzureFunctionsAuthenticationIn,
QueryDict);
    AzureFunctionsResponse.GetResultAsText(ResponseTxt);
    exit(ResponseTxt);
end;
```


```
codeunit 50124 "MNB Check Item Availability"
{
    trigger OnRun()
    var
        TaskResults: Dictionary of [Text, Text];
        ItemNo: Code[20];
    begin
        ItemNo := Page.GetBackgroundParameters().Get('ItemNo');
        TaskResults.Add('ExternalInventory',
GetExternalInventoryAzureFunction (ItemNo));
        Page.SetBackgroundTaskResult(TaskResults);
    end;
}
```

AZURE BLOB STORAGE

Azure Blob Storage is a Microsoft cloud service designed for storing large amounts of unstructured data, such as text or binary data. It is ideal for serving images or documents. The service offers scalability, high availability, and security, making it suitable for a wide range of applications, from simple file storage to big data analytics.

The Azure Blob Storage Container can be created directly inside the Azure Portal. First new storage account needs to be created.

[Home](#) > [workshopsaladv_1718013030633](#) | [Overview](#) >

**workshopsaladv**
Storage account

[Upload](#) [Open in Explorer](#) [Delete](#) [Move](#)

Overview

Activity log

Tags

Diagnose and solve problems

Access Control (IAM)

Resource group ([move](#)) : [workshopsaladv](#)


Location : eastus

Primary/Secondary Loc... : Primary: East US, Secondary: West US

Subscription ([move](#)) : [Visual Studio Enterprise Subscription](#)

The new container needs to be added to the storage account.

[Home](#) > [workshopsaladv_1718013030633](#) | [Overview](#) > [workshopsaladv](#)

**workshopsaladv** | Containers
Storage account

[+ Container](#) [Change access level](#) [Restore c...](#)

Overview

Activity log

Tags

Diagnose and solve problems

Access Control (IAM)

Data migration

Events

Storage browser

Storage Mover

Data storage

Containers

File shares

Queues

Search containers by prefix

Name
<input type="checkbox"/> \$logs
<input type="checkbox"/> workshops

To connect to the Azure Blob Storage account you can use the API key and Storage Account Name. Both can be found under the Access Keys section.

The screenshot shows the Microsoft Azure portal interface. At the top, there's a blue header with the Microsoft Azure logo and a search bar. Below the header, the breadcrumb navigation shows 'Home > workshopsaladv_1718013030633 | Overview > workshopsaladv'. The main heading is 'workshopsaladv | Access keys' with a star icon and a dropdown menu. Below the heading, there's a search bar and a navigation pane on the left with various options like Overview, Activity log, Tags, Diagnose and solve problems, Access Control (IAM), Data migration, Events, Storage browser, Storage Mover, Data storage, Containers, File shares, Queues, Tables, Security + networking, Networking, Front Door and CDN, Access keys (selected), and Shared access signature. The main content area shows the 'Access keys' section for the storage account 'workshopsaladv'. It includes a 'Set rotation reminder' button, a 'Refresh' button, and a 'Give feedback' button. Below these, there's a text block explaining access keys and a link to 'Learn more about managing storage account access keys'. The 'Storage account name' is 'workshopsaladv'. There are two keys listed: 'key1' and 'key2'. Each key has a 'Rotate key' button, a 'Last rotated' date (10.06.2024 (0 days ago)), a 'Key' field (masked with dots), and a 'Show' button. Below each key is a 'Connection string' field (masked with dots) and a 'Show' button.

TASK: CONNECT TO AZURE BLOB STORAGE

A customer asked to be able to send files to Azure Blob Storage directly from Business Central. The first step is to connect to Azure Blob Storage from Business Central.

1. Create a new table **MNB ABS Integration Setup** and add fields to connect to Azure Blob Storage: Azure Account Name, Container Name, Storage Key. Only for training purposes, the Storage Key will be stored directly inside the database however you can make it more secure in your example using isolated storage.
2. Create a new page **MNB ABS Integration Setup** with all fields from the table above.

3. Create a new page **MNB Azure Containers** that the source table is the **ABS Container** table.

Add to the page field Name.



HINT

The table is part of the SystemApp however there are no pages for the table since for each implementation of the functionality developers may need different user experiences.

4. Create a new codeunit **MNB Azure Blob Storage**
5. Add new function **ListContainers**. In the function get the **MNB ABS Integration Setup** record and make an **Authorization** using Interface "Storage Service Authorization". Then use the ABS Container Client codeunit and page created in the previous step to show all containers.
6. Save the value with the container name to the table
7. Add action to page **MNB ABS Integration Setup** to retrieve the list of Azure Blob Storage containers



SOLUTION

```
table 50102 "MNB ABS Integration Setup"
{
    Caption = 'Integration Setup';
    DataClassification = SystemMetadata;
    fields
    {
        field(1; "Primary Key"; Code[10])
        {
            Caption = 'Primary Key';
        }

        field(2; "Container Name"; Text[250])
        {
            Caption = 'Container Name';
        }

        field(3; "Azure Account Name"; Text[250])
        {
            Caption = 'Azure Account Name';
        }

        field(4; "Storage Key"; Text[1024])
    }
}
```

```

    {
        Caption = 'Storage Key';
        ExtendedDatatype = Masked;
    }
}

keys
{
    key(PK; "Primary Key")
    {
        Clustered = true;
    }
}

var
    RecordHasBeenRead: Boolean;

procedure GetRecordOnce()
begin
    if RecordHasBeenRead then
        exit;
    Get();
    RecordHasBeenRead := true;
end;

procedure InsertIfNotExists()
begin
    Reset();
    if not Get() then begin
        Init();
        Insert(true);
    end;
end;
}

```

```

page 50110 "MNB ABS Integration Setup"
{

    PageType = Card;
    SourceTable = "MNB ABS Integration Setup";
    Caption = 'Integration Setup';
    InsertAllowed = false;
    DeleteAllowed = false;
}

```

```

UsageCategory = Administration;
ApplicationArea = All;

layout
{
    area(content)
    {
        group(AzureBlobStorage)
        {
            Caption = 'Azure Blob Storage';
            field("Azure Account Name"; Rec."Azure Account Name")
            {
                Tooltip = 'Specifies the name of the Azure Blob
Storage account. The name is unique within the Azure environment and is
used to access the storage account.';
            }
            field("Container Name"; Rec."Container Name")
            {
                Tooltip = 'Specifies the name of the container in
the Azure Blob Storage account. The container is used to store the files
that are used by the integration service.';
                Editable = false;
            }
            field("Storage Key"; Rec."Storage Key")
            {
                Tooltip = 'Specifies the key that is used to access
the Azure Blob Storage account. The key is used to authenticate the user
and to access the storage account. The key is unique within the Azure
environment and is used to access the storage account. The key is used
to authenticate the user and to access the storage account. The key is
unique within the Azure environment and is used to access the storage
account.';
            }
        }
    }
}

trigger OnOpenPage()
begin
    Rec.InsertIfNotExists();
end;
}

```

```

page 50108 "MNB Azure Containers"
{
    Caption = 'Azure Containers';
    PageType = List;
    SourceTable = "ABS Container";
    UsageCategory = None;
    Editable = false;
    InsertAllowed = false;
    DeleteAllowed = false;
    ModifyAllowed = false;

    layout
    {
        area(content)
        {
            repeater(General)
            {
                field(Name; Rec.Name)
                {
                    ApplicationArea = All;
                    ToolTip = 'Specifies the value of the Name field.';
                }
            }
        }
    }
}

```

```

internal procedure ListContainers(): Text
var
    IntegrationSetup: Record "MNB ABS Integration Setup";
    ABSContainer: Record "ABS Container";
    ABSContainerClient: Codeunit "ABS Container Client";
    StorageServiceAuthorization: Codeunit "Storage Service
Authorization";
    Authorization: Interface "Storage Service Authorization";
begin
    IntegrationSetup.Get();
    Authorization :=
StorageServiceAuthorization.CreateSharedKey(IntegrationSetup."Storage
Key");

    ABSContainerClient.Initialize(IntegrationSetup."Azure Account
Name", Authorization);
    ABSContainerClient.ListContainers(ABSContainer);

```



```

        if Page.RunModal(Page::"MNB Azure Containers", ABSContainer) =
Action::LookupOK then
            exit(ABSContainer.Name);
        end;

```

```

action(ListContainers)
{
    Caption = 'List Containers';
    Image = ShowList;
    ToolTip = 'Show list of containers';
    ApplicationArea = All;
    trigger OnAction()
    var
        MNBAzureBlobStorage: Codeunit "MNB Azure Blob Storage";
    begin
        Rec."Container Name" := MNBAzureBlobStorage.ListContainers();
    end;
}

```



TASK: ADD ACTION TO SEND FILES TO AZURE BLOB STORAGE

1. Create a new function **ImportFileToAzureBlobStorage** in **MNB Azure Blob Storage** codeunit
2. In the function get the **MNB ABS Integration Setup** record and make an **Authorization** using the Interface "Storage Service Authorization"
3. Using the codeunit **ABS Blob Client** initialize the connection and then use the function **PutBlobBlockBlobStream** to upload the file to the instream
4. Add action to UploadFileIntoStream and run procedure to import the file to Azure Blob Storage



SOLUTION

```

procedure ImportFileToAzureBlobStorage(FileName: Text; var IStream:
InStream)
var
    ABSBlobClient: Codeunit "ABS Blob Client";
    StorageServiceAuthorization: Codeunit "Storage Service
Authorization";

```

```

    Authorization: Interface "Storage Service Authorization";
begin
    IntegrationSetup.Get();
    Authorization :=
StorageServiceAuthorization.CreateSharedKey(IntegrationSetup."Storage
Key");
    ABSBlobClient.Initialize(IntegrationSetup."Azure Account Name",
IntegrationSetup."Container Name", Authorization);
    ABSBlobClient.PutBlobBlockBlobStream(FileName, IStream);
end;

```

```

action(ImportFileToAzureBlobStorage)
{
    Caption = 'Import File';
    Image = Import;
    ToolTip = 'Import file to Azure Blob Storage';
    ApplicationArea = All;
    trigger OnAction()
    var
        MNBAzureBlobStorage: Codeunit "MNB Azure Blob Storage";
        IStream: InStream;
        FileName: Text;
    begin
        UploadIntoStream('', '', '', FileName, IStream);
        MNBAzureBlobStorage.ImportFileToAzureBlobStorage(FileName,
IStream);
    end;
}

```



TASK: ADD ACTION TO DOWNLOAD FILES FROM AZURE BLOB STORAGE

1. Create a new page **MNB Container Content** that the source table is **ABS Container Content** table. Add to the page field Name and Content Type.



HINT

The table is part of the SystemApp however there are no pages for the table since for each implementation of the functionality developers may need different user experiences.

2. Create a new function **DownloadFileFromAzureBlobStorage** in **MNB Azure Blob Storage** codeunit
3. In the function get the **MNB ABS Integration Setup** record and make an **Authorization** using the Interface "Storage Service Authorization"
4. Using the codeunit **ABS Blob Client** initialize the connection and then use function **ListBlobs** to list all blobs in the Azure Blob Storage Container. Open a list of all blobs using the created page **MNB Container Content**. If the user chooses the blob then use a function from the codeunit **ABS Blob Client - GetBlobAsFile**.
5. Add action to view all blobs and download the blob from the container



SOLUTION

```

page 50108 "MNB Azure Containers"
{
    Caption = 'Azure Containers';
    PageType = List;
    SourceTable = "ABS Container";
    UsageCategory = None;
    Editable = false;
    InsertAllowed = false;
    DeleteAllowed = false;
    ModifyAllowed = false;

    layout
    {
        {
            area(content)
            {
                repeater(General)
                {
                    field(Name; Rec.Name)
                    {
                        ApplicationArea = All;
                        ToolTip = 'Specifies the value of the Name field.';
                    }
                }
            }
        }
    }
}

```

```

internal procedure DownloadFileFromAzureBlobStorage()
var
    ABSContainerContent: Record "ABS Container Content";
    ABSBlobClient: Codeunit "ABS Blob Client";
    StorageServiceAuthorization: Codeunit "Storage Service
Authorization";
    Authorization: Interface "Storage Service Authorization";
begin
    IntegrationSetup.Get();
    IntegrationSetup.TestField("Container Name");
    Authorization :=
StorageServiceAuthorization.CreateSharedKey(IntegrationSetup."Storage
Key");
    ABSBlobClient.Initialize(IntegrationSetup."Azure Account Name",
IntegrationSetup."Container Name", Authorization);
    ABSBlobClient.ListBlobs(ABSContainerContent);
    if Page.RunModal(Page::"MNB Container Content", ABSContainerContent)
= Action::LookupOK then
        ABSBlobClient.GetBlobAsFile(ABSContainerContent.Name);
end;

```

```

action(DownloadFileFromAzureBlobStorage)
{
    Caption = 'Download File';
    Image = Import;
    ToolTip = 'Download file from Azure Blob Storage';
    ApplicationArea = All;
    trigger OnAction()
    var
        MNBAzureBlobStorage: Codeunit "MNB Azure Blob Storage";
    begin
        MNBAzureBlobStorage.DownloadFileFromAzureBlobStorage();
    end;
}

```

CHAPTER SUMMARY

- ✓ In this chapter, you learn about the SystemApp modules.
- ✓ You developed functionality to retrieve data from Azure Function
- ✓ You developed integration with files stored on Azure Blob Storage

LAST WORD

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.