Lab A: Lab 01 - Develop an Extension with C/SIDE

Scenario

We want to create some simple "Item Classification" functionality on a default database of Microsoft Dynamics NAV. For example, Items that get sold very often are of classification "A", while Items that get sold less, is category "D".

These classifications should be configurable, and we should have an action on the Item List that calculates the current classification of the Item. This Action should be implemented by an architecture that enables extendibility and decoupling of the method.

We will implement a notification as a warning whenever an Item is being use on the Sales Line, notifying the user of its classification, and enabling him to drill down into the Item and its Ledger Entries to provide more information about the Item and its classification.

To configure the App, we will create a Wizard. This Wizard will provide new default data if there is no data. We will make sure this Wizard is also available in the "Assisted Setup" functionality of Microsoft Dynamics NAV.

Last but not least, we will make sure our Extension is available in the Application Areas "basic" and "suite".

Objectives

- Learn how to
  - set up a new Development environment for your Extension
  - work with Notifications
  - create a Wizard
  - Implement Application Areas into your Extension
- Implement a decent design in your software that enables it to maintain, extend and decouple.
- Implement a SaaSified user experience by using notifications and Wizards

Exercise 1: Open the development environment

- Open the development environment
- Open the database "ItemClassification_DEV"

In this database (ItemClassification_DEV) you will do all the changes for this lab.

Exercise 2: Perform the necessary table changes

Exercise Scenario

The table changes that we need to do, are quite simple:

- A new table "Item Classification"

New field on the Item table

Task 1: Create Item Classification table

High Level Steps

1. Open the Object Designer of the ItemClassification_DEV database, and create the "Item Classification" table
2. Create a new page for this table called "Item Classifications"

Detailed Steps

1. Open the Object Designer of the ItemClassification_DEV database, and create the "Item Classification" table

The fields in this table are:

- Code (Code10)
- Description (Text50)
- Minimum Sales Count (Integer)

2. Create a new page for this table called "Item Classifications"

The page should be editable, and should also be the LookUp page of the table.

Task 2: Add the necessary field to the Item Table

High Level Steps

1. Add field "Item Classification Code" to the Item table
2. Put the field on the Item List and the Item Card

Detailed Steps

1. Add field "Item Classification Code" to the Item table

This field is a Code10, and has a table relation with the newly created table "Item Classification" and should not be editable. The value of this field will be filled automatically by the system.

2. Put the field on the Item List and the Item Card

Exercise 3: Implement the "CalcItemClassification" method

Exercise Scenario

The method needs to figure out how many times the Item has been sold. This number needs to be checked against the SalesCount-field in the "Item Classification" table, which gets you to the right "Item Classification Code".

Task 1: Create the Codeunit with the necessary business logic

High Level Steps

1. Create a typical Method codeunit
2. Create the local function "GetItemSalesCount"
3. Create the local function "GetItemClassificationCode"
4. Now finish the Codeunit so that you can call the "CalcItemClassification" function

Detailed Steps

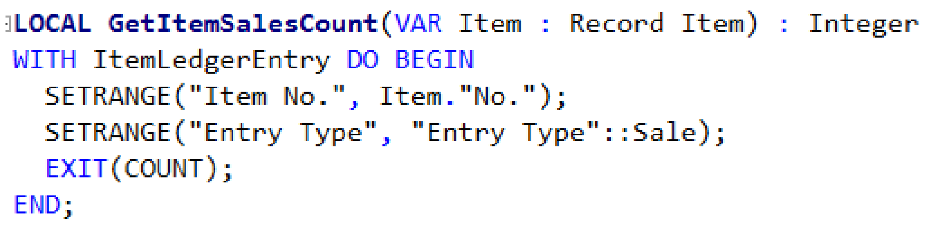
1. Create a typical Method codeunit

- Create a new codeunit "Calc. Item Classification Meth"
- Create a global function in the codeunit "CalcItemClassification"

Create a local function in that codeunit "DoCalcItemClassification"

2. Create the local function "GetItemSalesCount"

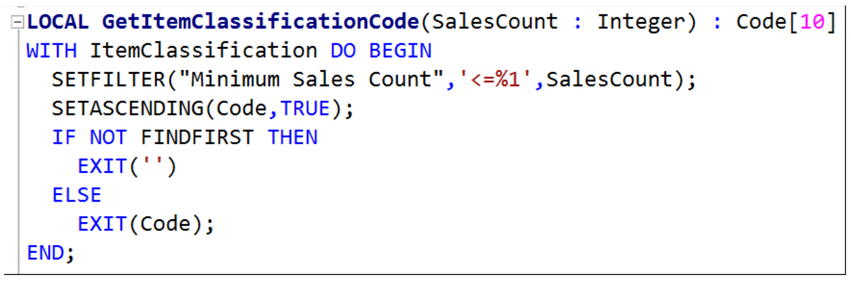
The business logic should be something like this:



```
LOCAL GetItemSalesCount(VAR Item : Record Item) : Integer
WITH ItemLedgerEntry DO BEGIN
  SETRANGE("Item No.", Item."No.");
  SETRANGE("Entry Type", "Entry Type"::Sale);
  EXIT(COUNT);
END;
```

3. Create the local function "GetItemClassificationCode"

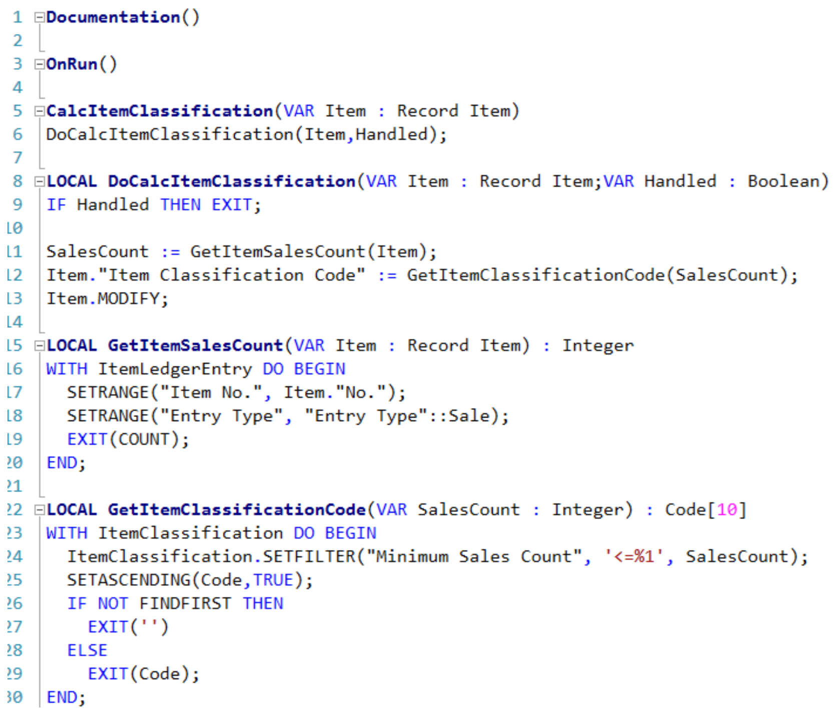
Which should look like this:



```
LOCAL GetItemClassificationCode(SalesCount : Integer) : Code[10]
WITH ItemClassification DO BEGIN
  SETFILTER("Minimum Sales Count",'<=%1',SalesCount);
  SETASCENDING(Code,TRUE);
  IF NOT FINDFIRST THEN
    EXIT('')
  ELSE
    EXIT(Code);
END;
```

4. Now finish the Codeunit so that you can call the "CalcItemClassification" function

The resulting codeunit should now look like this:



```
Documentation()

OnRun()

CalcItemClassification(VAR Item : Record Item)
DoCalcItemClassification(Item,Handled);

LOCAL DoCalcItemClassification(VAR Item : Record Item;VAR Handled : Boolean)
IF Handled THEN EXIT;

SalesCount := GetItemSalesCount(Item);
Item."Item Classification Code" := GetItemClassificationCode(SalesCount);
Item.MODIFY;

LOCAL GetItemSalesCount(VAR Item : Record Item) : Integer
WITH ItemLedgerEntry DO BEGIN
  SETRANGE("Item No.", Item."No.");
  SETRANGE("Entry Type", "Entry Type"::Sale);
  EXIT(COUNT);
END;

LOCAL GetItemClassificationCode(VAR SalesCount : Integer) : Code[10]
WITH ItemClassification DO BEGIN
  ItemClassification.SETFILTER("Minimum Sales Count", '<=%1', SalesCount);
  SETASCENDING(Code,TRUE);
  IF NOT FINDFIRST THEN
    EXIT('')
  ELSE
    EXIT(Code);
END;
```

Task 2: Extend the codeunit with the necessary events to be able to decouple or extend the business logic.

High Level Steps

1. Add two Integration Events to the method

Implement the events with the "VAR Handled" pattern to be able to decouple or extend on this codeunit

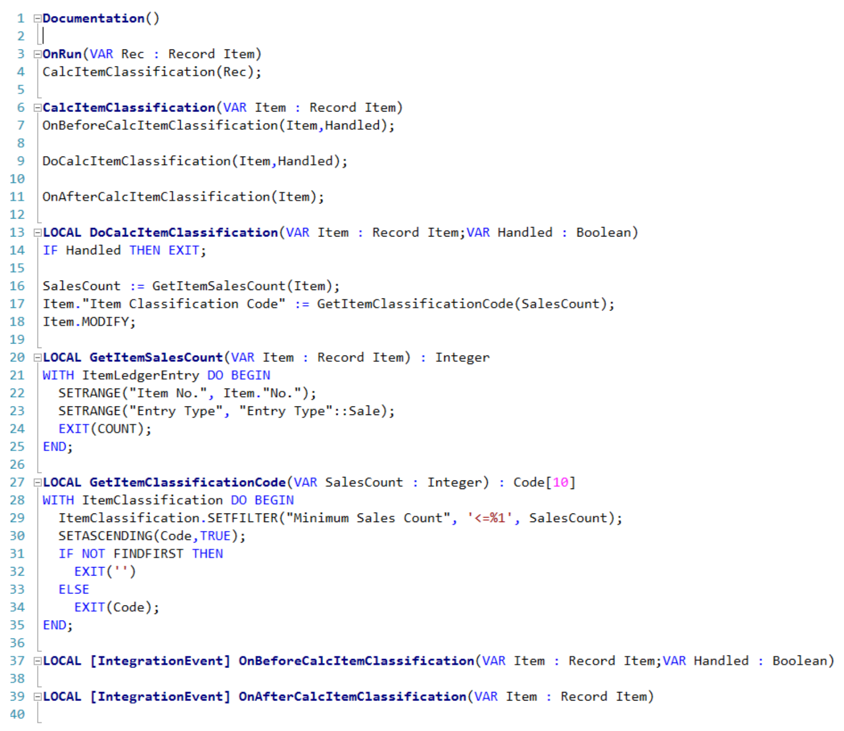
Detailed Steps

2. Add two Integration Events to the method

- OnBeforeCalcItemClassification
- OnAfterCalcItemClassification

Implement the events with the "VAR Handled" pattern to be able to decouple or extend on this codeunit

The result should look like this:



```
Documentation()

OnRun(VAR Rec : Record Item)
CalcItemClassification(Rec);

CalcItemClassification(VAR Item : Record Item)
OnBeforeCalcItemClassification(Item,Handled);

DoCalcItemClassification(Item,Handled);

OnAfterCalcItemClassification(Item);

LOCAL DoCalcItemClassification(VAR Item : Record Item;VAR Handled : Boolean)
IF Handled THEN EXIT;

SalesCount := GetItemSalesCount(Item);
Item."Item Classification Code" := GetItemClassificationCode(SalesCount);
Item.MODIFY;

LOCAL GetItemSalesCount(VAR Item : Record Item) : Integer
WITH ItemLedgerEntry DO BEGIN
  SETRANGE("Item No.", Item."No.");
  SETRANGE("Entry Type", "Entry Type"::Sale);
  EXIT(COUNT);
END;

LOCAL GetItemClassificationCode(VAR SalesCount : Integer) : Code[10]
WITH ItemClassification DO BEGIN
  ItemClassification.SETFILTER("Minimum Sales Count", '<=%1', SalesCount);
  SETASCENDING(Code,TRUE);
  IF NOT FINDFIRST THEN
    EXIT('')
  ELSE
    EXIT(Code);
END;

LOCAL [IntegrationEvent] OnBeforeCalcItemClassification(VAR Item : Record Item;VAR Handled : Boolean)

LOCAL [IntegrationEvent] OnAfterCalcItemClassification(VAR Item : Record Item)

```

Task 3: On the Item list, we need to be able to call the method

For extensions, this introduces a problem.

We are not able to add a function on a table from the default application, like the “Event Based Software Architecture” patters actually describes. So no way to be able to implement it so that we could call our method like “Item.CalcItemClassification”.

We can’t also not put any code on a page. So we can create the action, we just can’t put code there.

So that’s why we are going to add the action on the page, but call the code from a codeunit, where we add a subscription function to the action on the page.

High Level Steps

1. Create a new Action on the Item list
2. Call the method by using a subscription to the new Action

Detailed Steps

1. Create a new Action on the Item list

- Design the Item list
- View / Page Actions
- Add an Action “Calc. Item Classification”

- Properties:
  - Promoted=Yes;
  - PromotedCategory=Process;
  - PromotedIsBig=Yes;
  - Image=Calculate;

2. Call the method by using a subscription to the new Action

- Create a new codeunit "ItemClassification Subs"
- Create e new subscriber function "CalcItemClassification_OnActionItemList"
- Call the code

Task 4: Test the functionality

High Level Steps

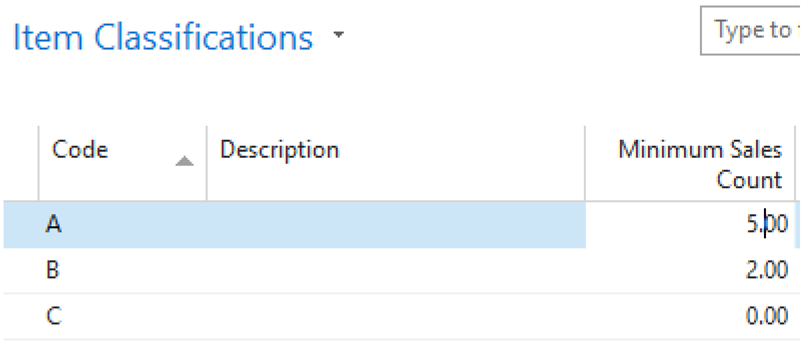
1. Open the Windows Client
2. Add data in the "Item Classifications"
3. Navigate to the Item List page

Detailed Steps

1. Open the Windows Client

Just run the "Item Classifications" page from the development environment.

2. Add data in the "Item Classifications"



Code	Description	Minimum Sales Count
A		5.00
B		2.00
C		0.00

3. Navigate to the Item List page

Find your new action and execute it for a few items.

Exercise 4: Implement notifications

Exercise Scenario

When using an Item (in Sales Line) that is marked in the "Item Classification" table as "Warning", the system should generate a notification to the user.

When the notification is shown, the user needs to be able to drill down to the Item Card, and the Item Ledger Entries.

Task 1: Create an extra field in the "Item Classification" table

High Level Steps

1. Add field "Warning" to the "Item Classification" table
2. Put created field on the "Item Classifications" page

Detailed Steps

1. Add field "Warning" to the "Item Classification" table
   - This field is a Boolean
2. Put created field on the "Item Classifications" page

Task 2: Create the Codeunit with the necessary business logic

High Level Steps

1. Create a typical Method Codeunit
2. Create the local function "GetItemClassWarningStatus"
3. Create the public function "HandleItemClassWarning_RunItemCard"
4. Create the public function "HandleItemClassWarning_RunItemLedgerEntries"
5. Create the local function "SendItemClassWarning"
6. Create the Event Subscriber "ShowItemClassWarning_OnValidateItemNoOnSalesLine"
7. Now finish the Codeunit so that you can call the "ShowItemClassWarning" function

Detailed Steps

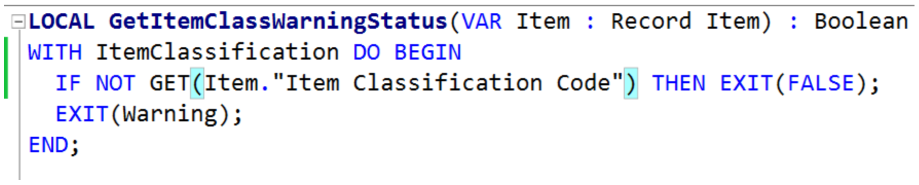
1. Create a typical Method Codeunit
   - Create a new Codeunit "ShowItemClassWarning Meth."
   - Create a global function in the Codeunit "ShowItemClassWarning"

Create a local function in that Codeunit "DoShowItemClassWarning"

2. Create the local function "GetItemClassWarningStatus"

This function will return whether the corresponding classification as marked as "warning" or not.

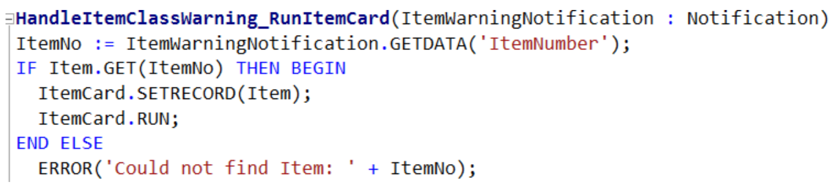
The business logic should be something like this:



```
LOCAL GetItemClassWarningStatus(VAR Item : Record Item) : Boolean
WITH ItemClassification DO BEGIN
  IF NOT GET(Item."Item Classification Code") THEN EXIT(FALSE);
  EXIT(Warning);
END;
```

3. Create the public function "HandleItemClassWarning_RunItemCard"

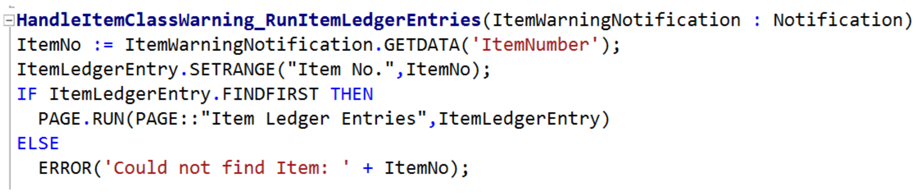
This function will handle the first action of the notification: running the Item Card.



```
HandleItemClassWarning_RunItemCard(ItemWarningNotification : Notification)
ItemNo := ItemWarningNotification.GETDATA('ItemNumber');
IF Item.GET(ItemNo) THEN BEGIN
  ItemCard.SETRECORD(Item);
  ItemCard.RUN;
END ELSE
  ERROR('Could not find Item: ' + ItemNo);
```

4. Create the public function "HandleItemClassWarning_RunItemLedgerEntries"

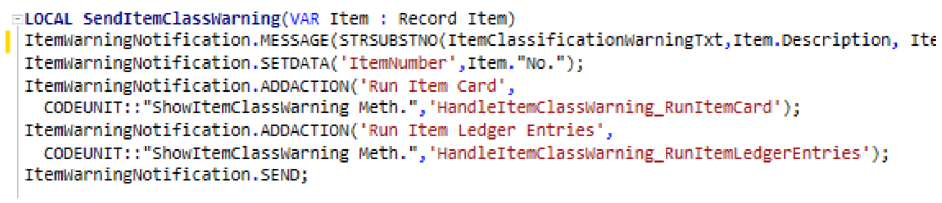
This function will handle the second action of the notification: running the Item Ledger Entries.



```
HandleItemClassWarning_RunItemLedgerEntries(ItemWarningNotification : Notification)
ItemNo := ItemWarningNotification.GETDATA('ItemNumber');
ItemLedgerEntry.SETRANGE("Item No.",ItemNo);
IF ItemLedgerEntry.FINDFIRST THEN
  PAGE.RUN(PAGE::"Item Ledger Entries",ItemLedgerEntry)
ELSE
  ERROR('Could not find Item: ' + ItemNo);
```

5. Create the local function "SendItemClassWarning"
   - ItemWarningNotification is of type "Notification"
   - ItemClassificationWarningTxt is a Text Constant with text: "Warning: Item %1 has an Item Classification %2!"

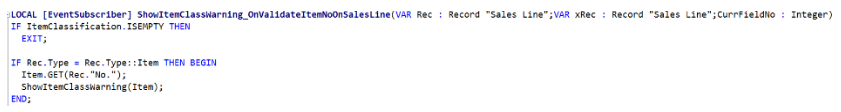
This is where we will send the notification.



```
LOCAL SendItemClassWarning(VAR Item : Record Item)
ItemWarningNotification.MESSAGE(STRSUBSTNO(ItemClassificationWarningTxt,Item.Description, Ite
ItemWarningNotification.SETDATA('ItemNumber',Item."No.");
ItemWarningNotification.ADDACTION('Run Item Card',
  CODEUNIT::"ShowItemClassWarning Meth.",'HandleItemClassWarning_RunItemCard');
ItemWarningNotification.ADDACTION('Run Item Ledger Entries',
  CODEUNIT::"ShowItemClassWarning Meth.",'HandleItemClassWarning_RunItemLedgerEntries');
ItemWarningNotification.SEND;
```

6. Create the Event Subscriber "ShowItemClassWarning_OnValidateItemNoOnSalesLine"

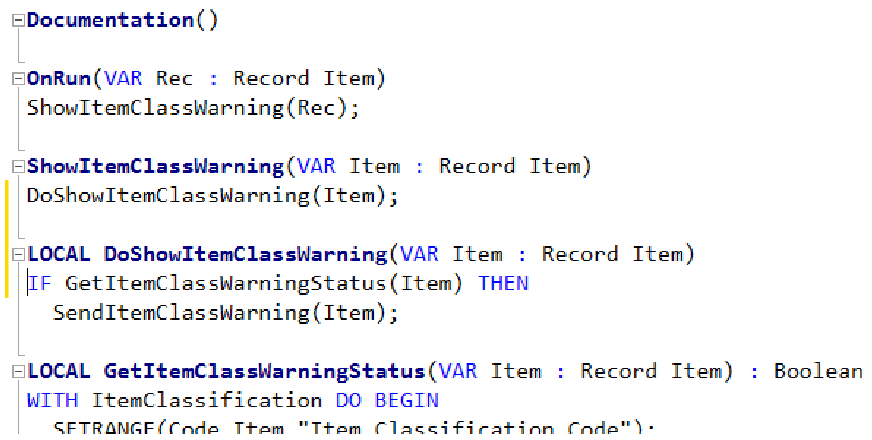
We need to call the notifications from somewhere by subscribing to the right event. This function is going to do that by subscribing to the “OnValidate” of the No.-field on the “Sales Line”



```
LOCAL [EventSubscriber] ShowItemClassWarning_OnValidateItemNoOnSalesLine(VAR Rec : Record "Sales Line";VAR xRec : Record "Sales Line";CurrFieldNo : Integer)
IF ItemClassification.ISEMPTY THEN
  EXIT;

IF Rec.Type = Rec.Type::Item THEN BEGIN
  Item.GET(Rec."No.");
  ShowItemClassWarning(Item);
END;
```

7. Now finish the Codeunit so that you can call the “ShowItemClassWarning” function

Basically connect the different pieces that we just have written in the main functions.



```
Documentation()

OnRun(VAR Rec : Record Item)
ShowItemClassWarning(Rec);

ShowItemClassWarning(VAR Item : Record Item)
DoShowItemClassWarning(Item);

LOCAL DoShowItemClassWarning(VAR Item : Record Item)
IF GetItemClassWarningStatus(Item) THEN
  SendItemClassWarning(Item);

LOCAL GetItemClassWarningStatus(VAR Item : Record Item) : Boolean
WITH ItemClassification DO BEGIN
  SETRANGE(Code,Item."Item Classification Code");
```

Task 3: Extend the Codeunit with the necessary events to be able to decouple or extend the business logic

High Level Steps

1. Add two Integration Events to the method
2. Implement the events with the “Handled VAR” pattern to be able to decouple or extend on this Codeunit

Detailed Steps

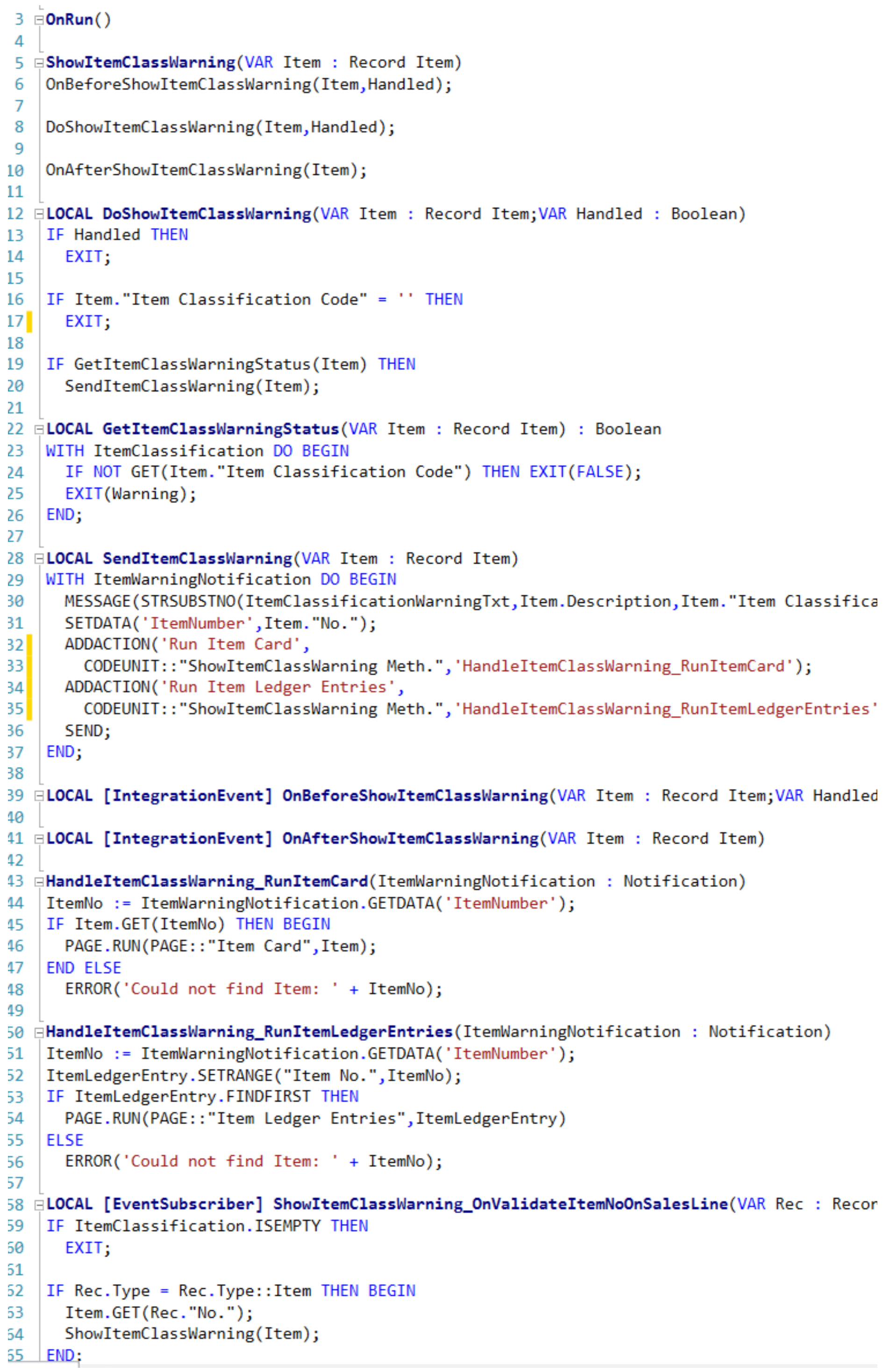
1. Add two Integration Events to the method

- OnBeforeShowItemClassWarning

OnAfterShowItemClassWarning

2. Implement the events with the “Handled VAR” pattern to be able to decouple or extend on this Codeunit

The result should look like this



```
OnRun()

ShowItemClassWarning(VAR Item : Record Item)
OnBeforeShowItemClassWarning(Item,Handled);

DoShowItemClassWarning(Item,Handled);

OnAfterShowItemClassWarning(Item);

LOCAL DoShowItemClassWarning(VAR Item : Record Item;VAR Handled : Boolean)
IF Handled THEN
  EXIT;

IF Item."Item Classification Code" = '' THEN
  EXIT;

IF GetItemClassWarningStatus(Item) THEN
  SendItemClassWarning(Item);

LOCAL GetItemClassWarningStatus(VAR Item : Record Item) : Boolean
WITH ItemClassification DO BEGIN
  IF NOT GET(Item."Item Classification Code") THEN EXIT(FALSE);
  EXIT(Warning);
END;

LOCAL SendItemClassWarning(VAR Item : Record Item)
WITH ItemWarningNotification DO BEGIN
  MESSAGE(STRSUBSTNO(ItemClassificationWarningTxt,Item.Description,Item."Item Classifica
  SETDATA('ItemNumber',Item."No.");
  ADDACTION('Run Item Card',
    CODEUNIT::"ShowItemClassWarning Meth.",'HandleItemClassWarning_RunItemCard');
  ADDACTION('Run Item Ledger Entries',
    CODEUNIT::"ShowItemClassWarning Meth.",'HandleItemClassWarning_RunItemLedgerEntries'
  SEND;
END;

LOCAL [IntegrationEvent] OnBeforeShowItemClassWarning(VAR Item : Record Item;VAR Handled

LOCAL [IntegrationEvent] OnAfterShowItemClassWarning(VAR Item : Record Item)

HandleItemClassWarning_RunItemCard(ItemWarningNotification : Notification)
ItemNo := ItemWarningNotification.GETDATA('ItemNumber');
IF Item.GET(ItemNo) THEN BEGIN
  PAGE.RUN(PAGE::"Item Card",Item);
END ELSE
  ERROR('Could not find Item: ' + ItemNo);

HandleItemClassWarning_RunItemLedgerEntries(ItemWarningNotification : Notification)
ItemNo := ItemWarningNotification.GETDATA('ItemNumber');
ItemLedgerEntry.SETRANGE("Item No.",ItemNo);
IF ItemLedgerEntry.FINDFIRST THEN
  PAGE.RUN(PAGE::"Item Ledger Entries",ItemLedgerEntry)
ELSE
  ERROR('Could not find Item: ' + ItemNo);

LOCAL [EventSubscriber] ShowItemClassWarning_OnValidateItemNoOnSalesLine(VAR Rec : Recor
IF ItemClassification.ISEMPTY THEN
  EXIT;

IF Rec.Type = Rec.Type::Item THEN BEGIN
  Item.GET(Rec."No.");
  ShowItemClassWarning(Item);
END;
```

Task 4: Test the functionality

High Level Steps

Open the Windows Client and test your code.

Detailed Steps

Open the Windows Client and test your code.

- Just run the "Item Classifications" page from the development environment.
- Make sure all classifications have "warning "to true.
- Use an Item with a calculated Classification in in a Sales line

When the notification is shown, try both available actions ("Run Item Card" and "Run Item Ledger Entries")

Exercise 5: Wizard for initial Setup

Exercise Scenario

We will create a Wizard page for the users to make it easier for them to initially create configuration data for the Extension.

The idea is again quite simple: if there is no data, the Wizard will notify that there is non, and foresee data for the user as "default data", which the user can change for himself.

If there IS data, there is no need for the Wizard.

Task 1: Create a new page

High Level Steps

Create a new page that will host the Wizard

Detailed Steps

Create a new page that will host the Wizard

- For the source table use "Item Classification"
- Choose option "Create a page using a wizard", and select NavigatePage.
- Save page as "Item Classification Wizard"

Task 2: Add steps to a wizard page

High Level Steps

1. Add two groups to the page.
2. Create global variables that will be used for setting Group visibility

3. Define Group visibility
4. Implement the first step
5. Implement the second step

Detailed Steps

1. Add two groups to the page.

Each group will represent one step in the Wizard:

- FirstStep

SecondStep.

2. Create global variables that will be used for setting Group visibility

- FirstStepVisible (Boolean)

SecondStepVisible (Boolean)

3. Define Group visibility

- Select Group
- View / Properties
- "Visible" property for the "FirstStep" is: "FirstStepVisible"

"Visible" property for the "SecondStep" is: "SecondStepVisible"

4. Implement the first step

- Add three subgroups to the first step.
- Add a global var: TableHasData (Boolean)
- Set properties of first subgroup:
    - InstructionalTextML: "Welcome!"
- Set properties of second subgroup:
    - Visible: TableHasData
    - InstructionalTextML: "There is already data in the Item Classification table."
- Set properties of third subgroup:
    - Visible: NOT TableHasData

InstructionalTextML: "There is no data in the Item Classification table, so we have provided some default values. Click Next and review the data."

5. Implement the second step

- Add subgroup to the group "SecondStep", name it "ItemClassificationData" and make it a repeater.
- Add every field from the underlying table to the repeater

Task 3: Implement the "InsertDefaultValue" logic for this Wizard

High Level Steps

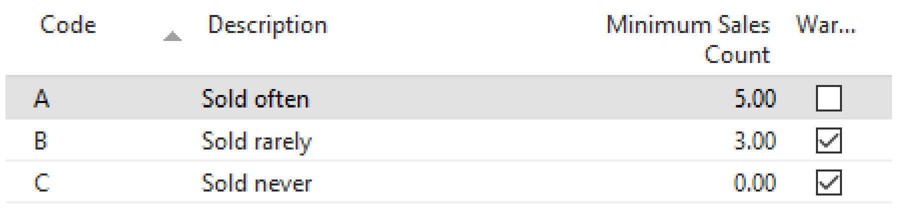
Create a new method codeunit that inserts three records in the "Item Classification" table.

Call this codeunit from a method on the table

Detailed Steps

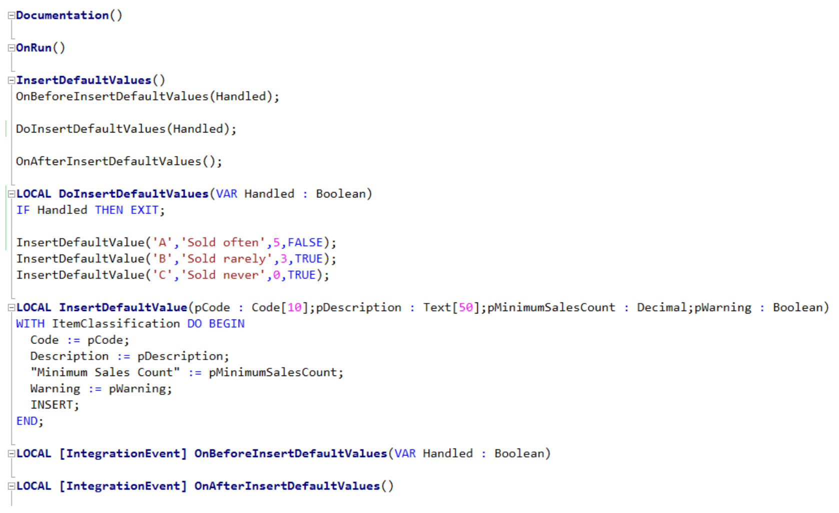
Create a new method codeunit that inserts three records in the "Item Classification" table.

- Create a typical method-codeunit with the name "InsertDefaultValues Meth."
- Implement the extensibility with OnBefore- and OnAfter-events
- Create a function that inserts one specific record
- Call that function three times to create these three records



Code	Description	Minimum Sales Count	War...
A	Sold often	5.00	☐
B	Sold rarely	3.00	☑
C	Sold never	0.00	☑

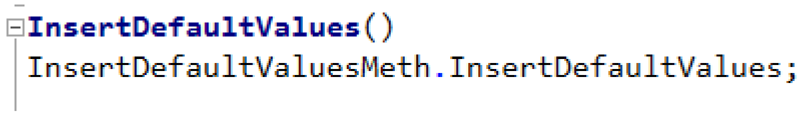
And this should be your codeunit:



```
Documentation()

OnRun()

InsertDefaultValues()
OnBeforeInsertDefaultValues(Handled);

DoInsertDefaultValues(Handled);

OnAfterInsertDefaultValues();

LOCAL DoInsertDefaultValues(VAR Handled : Boolean)
IF Handled THEN EXIT;

InsertDefaultValue('A','Sold often',5,FALSE);
InsertDefaultValue('B','Sold rarely',3,TRUE);
InsertDefaultValue('C','Sold never',0,TRUE);

LOCAL InsertDefaultValue(pCode : Code[10];pDescription : Text[50];pMinimumSalesCount : Decimal;pWarning : Boolean)
WITH ItemClassification DO BEGIN
  Code := pCode;
  Description := pDescription;
  "Minimum Sales Count" := pMinimumSalesCount;
  Warning := pWarning;
  INSERT;
END;

LOCAL [IntegrationEvent] OnBeforeInsertDefaultValues(VAR Handled : Boolean)

LOCAL [IntegrationEvent] OnAfterInsertDefaultValues()
```

Call this codeunit from a method on the table

- Create a new function "InsertDefaultValues" in table "Item Classification"
- Call the global function in the new codeunit



```
InsertDefaultValues()
InsertDefaultValuesMeth.InsertDefaultValues;
```

Task 4: Add Navigation to the Wizard

High Level Steps

1. Create Navigation Actions on the Item Classification Wizard
2. Create the local function "ResetControls"
3. Create the local function "EnableControls"
4. Create the local function "ShowFirstStep"
5. Create the local function "ShowSecondStep"
6. Create the local function "NextStep"
7. Create the local function "FinishAction"
8. Create navigation logic

Detailed Steps

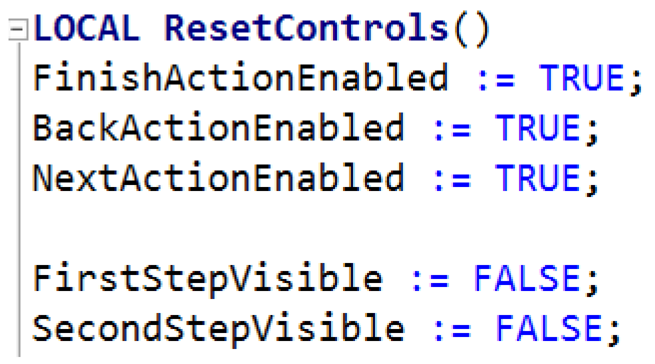
1. Create Navigation Actions on the Item Classification Wizard

- View / Page Actions
- Create "ActionContainer" with a SubType "ActionItems".
- Add an Action "ActionBack" with Caption: "Back"
- Add an Action " ActionNext" with Caption: "Next"
- Add an Action "ActionFinish" with Caption: "Finish"
- Create global variables that will be used for enabling navigation
    - Step (Option: "First,Second")
    - BackActionEnabled (Boolean)
    - NextActionEnabled (Boolean)
    - FinishActionEnabled (Boolean)
- Properties for the "ActionBack" are:
    - "Enabled": "BackActionEnabled"
    - "Image": "PreviousRecord"
    - "InFooterBar": "Yes"
- Properties for the "ActionNext" are:
    - "Enabled": "NextActionEnabled"
    - "Image": "NextRecord"
    - "InFooterBar": "Yes"

- Properties for the “ActionFinish” are:
    - “Enabled”: “FinishActionEnabled”
    - “Image”: “Approve”
    - “InFooterBar”: “Yes”

2. Create the local function “ResetControls”

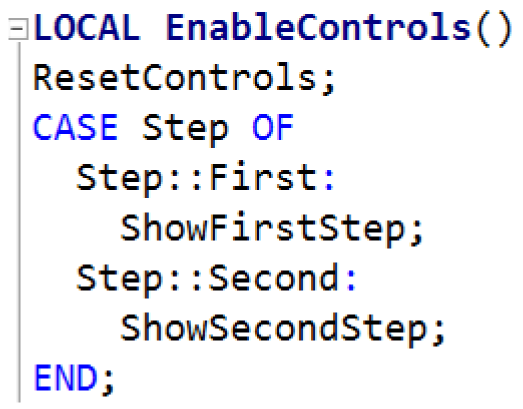
This function is going to reset all setup for the controls we have.



```
LOCAL ResetControls()
FinishActionEnabled := TRUE;
BackActionEnabled := TRUE;
NextActionEnabled := TRUE;

FirstStepVisible := FALSE;
SecondStepVisible := FALSE;
```

3. Create the local function “EnableControls”

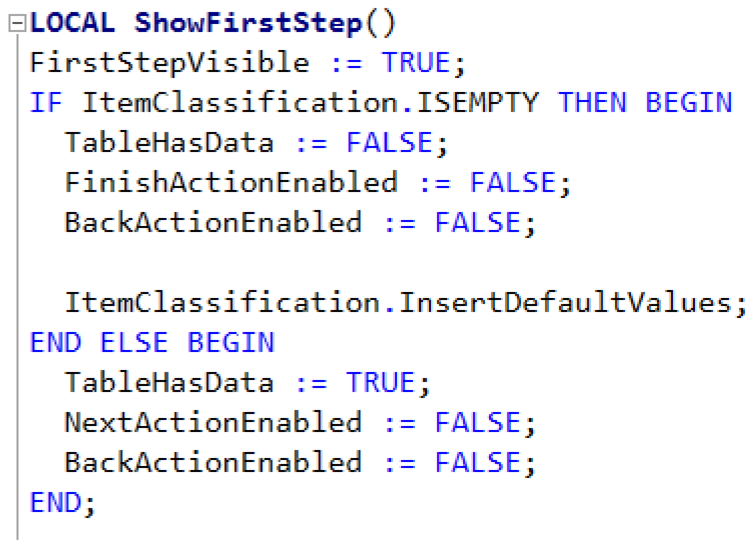
This function will enable the right controls depending on the step we’re in:



```
LOCAL EnableControls()
ResetControls;
CASE Step OF
  Step::First:
    ShowFirstStep;
  Step::Second:
    ShowSecondStep;
END;
```

4. Create the local function “ShowFirstStep”

This could be the logic.

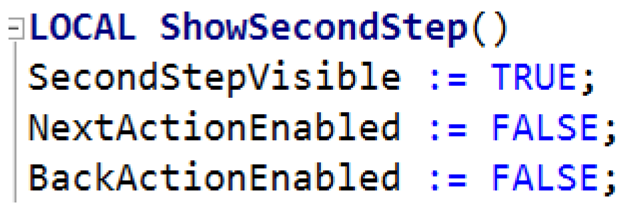


```
LOCAL ShowFirstStep()
FirstStepVisible := TRUE;
IF ItemClassification.ISEMPTY THEN BEGIN
  TableHasData := FALSE;
  FinishActionEnabled := FALSE;
  BackActionEnabled := FALSE;

  ItemClassification.InsertDefaultValues;
END ELSE BEGIN
  TableHasData := TRUE;
  NextActionEnabled := FALSE;
  BackActionEnabled := FALSE;
END;
```

You could debate whether it is a good thing to have our records created at this point. Basically it can – as part of the Wizard and the Wizard only.

5. Create the local function "ShowSecondStep"

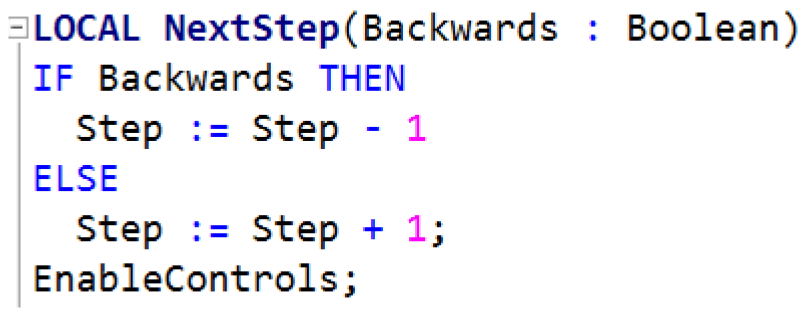
And this is what the second step looks like (showing the table):



```
LOCAL ShowSecondStep()
SecondStepVisible := TRUE;
NextActionEnabled := FALSE;
BackActionEnabled := FALSE;
```

6. Create the local function "NextStep"

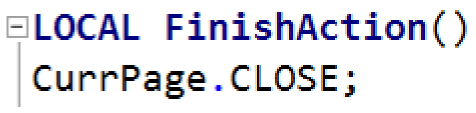
We also need to handle the steps:



```
LOCAL NextStep(Backwards : Boolean)
IF Backwards THEN
  Step := Step - 1
ELSE
  Step := Step + 1;
EnableControls;
```

7. Create the local function "FinishAction"

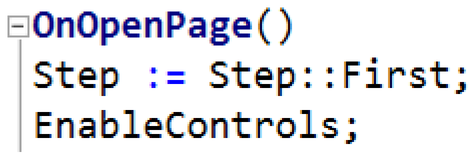
And we keep the "Finish" quite simple:



```
LOCAL FinishAction()
CurrPage.CLOSE;
```

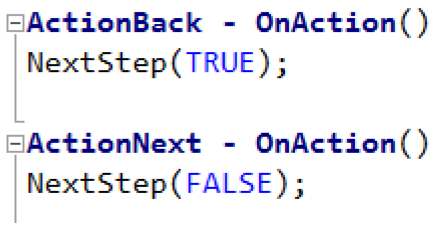
8. Create navigation logic

- On the "OnOpenPage" trigger, add this piece of code:



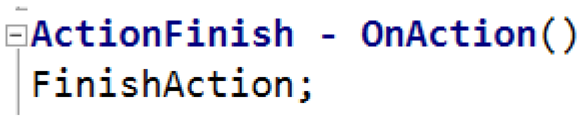
```
OnOpenPage()
Step := Step::First;
EnableControls;
```

- On "ActionBack" and "ActionNext", we need to call the NextStep:



```
ActionBack - OnAction()
NextStep(TRUE);

ActionNext - OnAction()
NextStep(FALSE);
```

- And on "ActionFinish", we need to call the "FinishAction" function:



```
ActionFinish - OnAction()
FinishAction;
```

Task 5: Add image header

We want to foresee the default header images that are visible in the Web Client (and Web client only).

High Level Steps

1. Create Groups to hold Media fields
2. Create the local function "LoadTopBanners"
3. Define Header appearance logic
4. Review the results

Detailed Steps

1. Create Groups to hold Media fields

- Create global variables:
    - "MediaRepositoryStandard" (Record – "Media Repository")

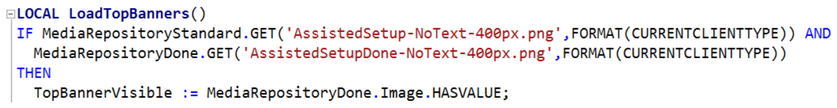
- "MediaRespositoryDone" (Record – "Media Repository")
- "TopBannerVisible" (Boolean)

- On the Wizard page, on top of all existing Groups create two new Groups
  - Add field to the first group: Type: "Field", SourceExpr: "MediaRepositoryStandard.Image"
  - Add field to the second group: Type: "Field", SourceExpr: "MediaRepositoryDone.Image"
- Set Group properties
- Properties for the first Group are:
  - "Visible": "TopBannerVisible AND NOT SecondStepVisible"
  - "Editable": "FALSE"
- Properties for the second Group are:
  - "Visible": "TopBannerVisible AND SecondStepVisible"

"Editable": "FALSE"

2. Create the local function "LoadTopBanners"

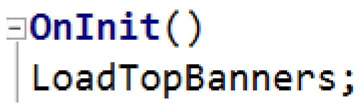
We need this code to get the right pictures out of the media repository. This code can be found on any Wizard page.



```
LOCAL LoadTopBanners()
IF MediaRepositoryStandard.GET('AssistedSetup-NoText-400px.png',FORMAT(CURRENTCLIENTTYPE)) AND
  MediaRepositoryDone.GET('AssistedSetupDone-NoText-400px.png',FORMAT(CURRENTCLIENTTYPE))
THEN
  TopBannerVisible := MediaRepositoryDone.Image.HASVALUE;
```

3. Define Header appearance logic

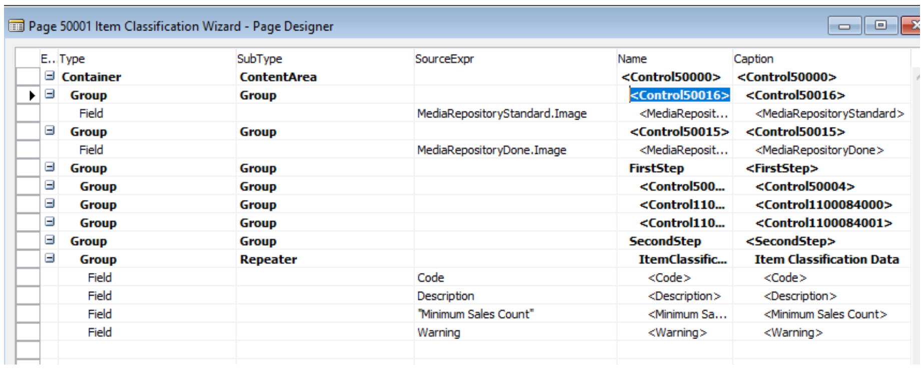
And to initialize the banners, we need to call the function "OnInit" of the page:



```
OnInit()
LoadTopBanners;
```

4. Review the results

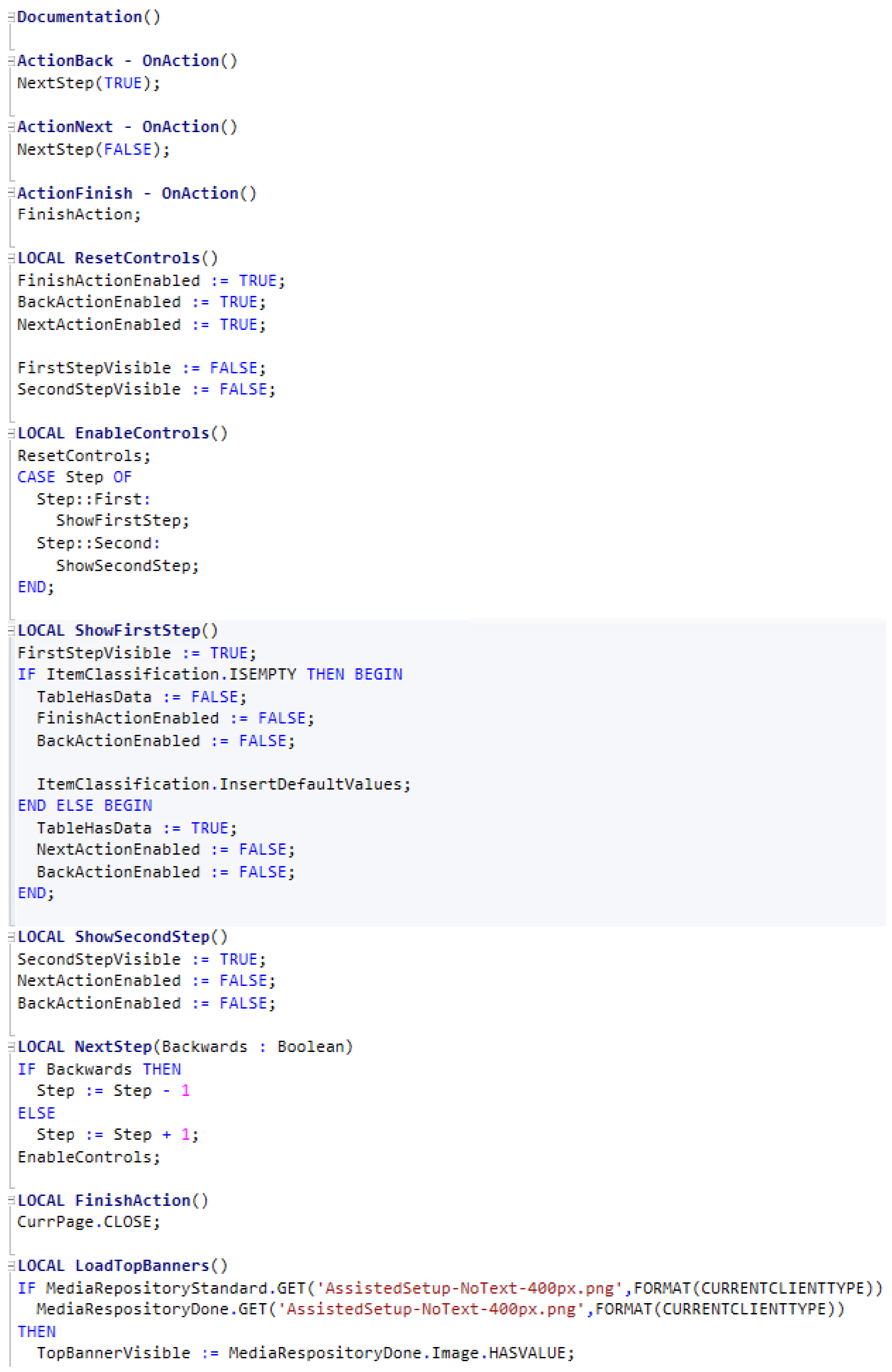
Finally, the Wizard page should look something like this



Page 50001 Item Classification Wizard - Page Designer

E..	Type	SubType	SourceExpr	Name	Caption
⊟	**Container**	**ContentArea**		**<Control50000>**	**<Control50000>**
⊟	**Group**	**Group**		**<Control50016>**	**<Control50016>**
	Field		MediaRepositoryStandard.Image	<MediaReposit…	<MediaRepositoryStandard>
⊟	**Group**	**Group**		**<Control50015>**	**<Control50015>**
	Field		MediaRepositoryDone.Image	<MediaReposit…	<MediaRepositoryDone>
⊟	**Group**	**Group**		**FirstStep**	**<FirstStep>**
⊟	**Group**	**Group**		**<Control500...**	**<Control50004>**
⊟	**Group**	**Group**		**<Control110...**	**<Control1100084000>**
⊟	**Group**	**Group**		**<Control110...**	**<Control1100084001>**
⊟	**Group**	**Group**		**SecondStep**	**<SecondStep>**
⊟	**Group**	**Repeater**		**ItemClassific...**	**Item Classification Data**
	Field		Code	<Code>	<Code>
	Field		Description	<Description>	<Description>
	Field		"Minimum Sales Count"	<Minimum Sa…	<Minimum Sales Count>
	Field		Warning	<Warning>	<Warning>

And the code should look something like this:



```
Documentation()

ActionBack - OnAction()
NextStep(TRUE);

ActionNext - OnAction()
NextStep(FALSE);

ActionFinish - OnAction()
FinishAction;

LOCAL ResetControls()
FinishActionEnabled := TRUE;
BackActionEnabled := TRUE;
NextActionEnabled := TRUE;

FirstStepVisible := FALSE;
SecondStepVisible := FALSE;

LOCAL EnableControls()
ResetControls;
CASE Step OF
  Step::First:
    ShowFirstStep;
  Step::Second:
    ShowSecondStep;
END;

LOCAL ShowFirstStep()
FirstStepVisible := TRUE;
IF ItemClassification.ISEMPTY THEN BEGIN
  TableHasData := FALSE;
  FinishActionEnabled := FALSE;
  BackActionEnabled := FALSE;

  ItemClassification.InsertDefaultValues;
END ELSE BEGIN
  TableHasData := TRUE;
  NextActionEnabled := FALSE;
  BackActionEnabled := FALSE;
END;

LOCAL ShowSecondStep()
SecondStepVisible := TRUE;
NextActionEnabled := FALSE;
BackActionEnabled := FALSE;

LOCAL NextStep(Backwards : Boolean)
IF Backwards THEN
  Step := Step - 1
ELSE
  Step := Step + 1;
EnableControls;

LOCAL FinishAction()
CurrPage.CLOSE;

LOCAL LoadTopBanners()
IF MediaRepositoryStandard.GET('AssistedSetup-NoText-400px.png',FORMAT(CURRENTCLIENTTYPE))
  MediaRespositoryDone.GET('AssistedSetup-NoText-400px.png',FORMAT(CURRENTCLIENTTYPE))
THEN
  TopBannerVisible := MediaRespositoryDone.Image.HASVALUE;
```

Task 6: Test the functionality

High Level Steps

Open the Windows Client and test your code.

Detailed Steps

Open the Windows Client and test your code.

- Run the Item Classifications page
- Delete all records
- Run the Wizard page
- Step through all the steps and look at the buttons (enabled/disabled)

Verify the data by calculating a few classifications in the Item list.

Exercise 6: Assisted Setup

Exercise Scenario

We will add the wizard that we created in the previous Exercise to the Assisted Setup.

This is basically being done by subscribing to functions of the "Aggregated Assisted Setup".

Task 1: Use the events on the "Aggregated Assisted Setup" to add your own set to the functionality.

High Level Steps

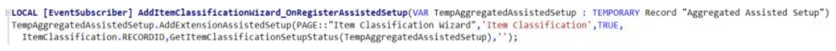
1. Create the Event Subscriber "AddItemClassificationWizard_OnRegisterAssistedSetup"
2. Create the Event Subscriber "UpdateItemClassificationSetupStatus_OnUpdateAssistedSetupStatus "
3. Create the local function "GetItemClassificationSetupStatus"

Detailed Steps

1. Create the Event Subscriber "AddItemClassificationWizard_OnRegisterAssistedSetup"

- Open the Codeunit "ItemClassification Subs"
- Create the Event Subscriber "AddItemClassificationWizard_OnRegisterAssistedSetup"

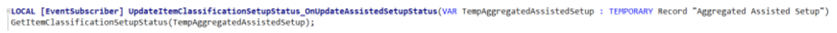
The business logic should be something like this:



```
LOCAL [EventSubscriber] AddItemClassificationWizard_OnRegisterAssistedSetup(VAR TempAggregatedAssistedSetup : TEMPORARY Record "Aggregated Assisted Setup")
TempAggregatedAssistedSetup.AddExtensionAssistedSetup(PAGE::"Item Classification Wizard",'Item Classification',TRUE,
  ItemClassification.RECORDID,GetItemClassificationSetupStatus(TempAggregatedAssistedSetup),'');
```

2. Create the Event Subscriber "UpdateItemClassificationSetupStatus_OnUpdateAssistedSetupStatus "

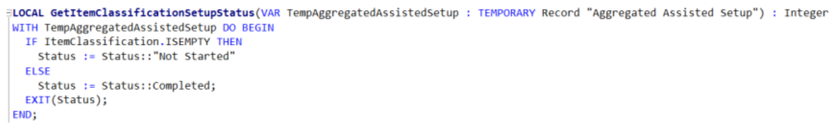
This event will be called to check the status of the particular setup.



```
LOCAL [EventSubscriber] UpdateItemClassificationSetupStatus_OnUpdateAssistedSetupStatus(VAR TempAggregatedAssistedSetup : TEMPORARY Record "Aggregated Assisted Setup")
GetItemClassificationSetupStatus(TempAggregatedAssistedSetup);
```

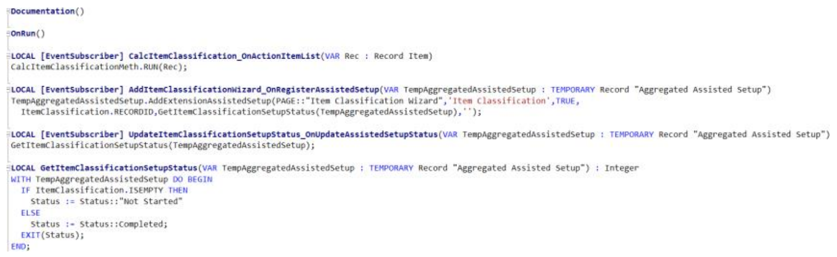
3. Create the local function “GetItemClassificationSetupStatus”

To get the status of our setup, let’s implement that in a local function like this:



```
LOCAL GetItemClassificationSetupStatus(VAR TempAggregatedAssistedSetup : TEMPORARY Record "Aggregated Assisted Setup") : Integer
WITH TempAggregatedAssistedSetup DO BEGIN
  IF ItemClassification.ISEMPTY THEN
    Status := Status::"Not Started"
  ELSE
    Status := Status::Completed;
  EXIT(Status);
END;
```

All-in-all, the resulting codeunit looks like this:



```
Documentation()

OnRun()

LOCAL [EventSubscriber] CalcItemClassification_OnActionItemList(VAR Rec : Record Item)
CalcItemClassificationMeth.RUN(Rec);

LOCAL [EventSubscriber] AddItemClassificationWizard_OnRegisterAssistedSetup(VAR TempAggregatedAssistedSetup : TEMPORARY Record "Aggregated Assisted Setup")
TempAggregatedAssistedSetup.AddExtensionAssistedSetup(PAGE::"Item Classification Wizard",'Item Classification',TRUE,
  ItemClassification.RECORDID,GetItemClassificationSetupStatus(TempAggregatedAssistedSetup),'');

LOCAL [EventSubscriber] UpdateItemClassificationSetupStatus_OnUpdateAssistedSetupStatus(VAR TempAggregatedAssistedSetup : TEMPORARY Record "Aggregated Assisted Setup")
GetItemClassificationSetupStatus(TempAggregatedAssistedSetup);

LOCAL GetItemClassificationSetupStatus(VAR TempAggregatedAssistedSetup : TEMPORARY Record "Aggregated Assisted Setup") : Integer
WITH TempAggregatedAssistedSetup DO BEGIN
  IF ItemClassification.ISEMPTY THEN
    Status := Status::"Not Started"
  ELSE
    Status := Status::Completed;
  EXIT(Status);
END;
```

Task 2: Test the functionality

High Level Steps

Open the Windows Client and test your code.

Detailed Steps

Open the Windows Client and test your code.

- Run the Item Classifications page
- Delete all records
- Run the “Assisted Setup” page from the development environment.
- Verify that your new setup has been added to the page.
- Click it
- Run through it
- Verify that the status is completed.

Exercise 7: Application Area

Exercise Scenario

Make sure our app is working under these Application Areas: #Basic,#Suite

Task 1: Set up Application Area on Item List and Item Card

High Level Steps

Set Application Area property for the "Item Classification Code"

Detailed Steps

Set Application Area property for the "Item Classification Code"

- Open Item List page
- Select "Item Classification Code"
- View / Properties
- Set property “ApplicationArea” to “#Basic,#Suite”

Do the same for the Item Card

Task 2: Set up Application Area on Item Classification

High Level Steps

Set “Application Area” property for all the fields on the “Item Classifications” page

Detailed Steps

Set “Application Area” property for all the fields on the “Item Classifications” page

For each of the fields on the “Item Classifications” page:

- Select field
- View / Properties

Set property “ApplicationArea” to “#Basic,#Suite”

Task 3: Set up Application Area on Item Classification Wizard

High Level Steps

Set “Application Area” property for all the fields on the “Item Classification Wizard” page

Detailed Steps

Set “Application Area” property for all the fields on the “Item Classification Wizard” page

For each of the fields on the “Item Classification Wizard” page:

- Select field
- View / Properties
- Set property “ApplicationArea” to “#Basic,#Suite”

Exercise 8: Create Notification for unconfigured Extension.

Exercise Scenario

Create Notification for when the Extension is just installed, but the user hasn’t configured it through the assisted setup.

The notification should show a warning like: “You haven’t set up the “Item Classification” setup yet. You can do that here” (with “here” a link to the wizard). Obviously, it should only show that if the Extension isn’t set up yet (no data in the Item Classification table).

The Notification should get be shown at the these events:

- When you installed the Extension from the Extension Manager
- When you open the Item List or Item Card
- When you use any Item on the Sales Line

Task 1: Try to implement this Exercise yourself without us providing any steps for it.

High Level Steps

1. Create a Method Codeunit
2. Find and subscribe to the right events to call your new method

Detailed Steps

1. Create a Method Codeunit
2. Find and subscribe to the right events to call your new method

Exercise 9: Publish and test as an Extension

Exercise Scenario

Only if time permits, try to publish all this as an Extension with waldo’s PowerShell Scripts.

Task 1: Use the scripts in the folder “PSScripts/NAV Extensions“

High Level Steps

1. Find the scripts in the downloaded folder
2. Read the .md file and try to figure out yourself on how to execute this lab

Detailed Steps

1. Find the scripts in the downloaded folder
2. Read the .md file and try to figure out yourself on how to execute this lab

Results

You should have created a new test instance first (ItemClassification_TEST) where you can test your app.

You should also have executed the “Build&Deploy” script that builds the app from your DEV environment, and published it to your newly created test environment.

Lab 02 - Transform the Extension into AL code with VS Code

Scenario

We will develop the same scenario as we did in the first lab.

The reason why we will do this, is so you can very good compare what has been change, and what needs to be done to accomplish the same as in the classic development environment.

Objectives

The objectives of this lab are the same as in the first lab, only applied on Extensions v2 :

- Learn how to
    - set up a new Development environment for your Extension
    - work with Notifications
    - create a Wizard
    - Implement Application Areas into your Extension
- Implement a decent design in your software that enables it to maintain, extend and decouple.
- Implement a SaaSified user experience by using notifications and Wizards

Exercise 1: Prepare your development environment

Exercise Scenario

A project in VSCode is actually just a folder on your filesystem, basically because it is all "file based". We will be writing al files, and turn that into an extension.

Task 1: Create a new VSCode project

High Level Steps

1. Open VSCode
2. Start the new project
3. Remove Unnecessary items
4. Add information to the app.json

Detailed Steps

1. Open VSCode

Double click on "Visual Studio Code" on the desktop or search for it in the Start menu

2. Start the new project

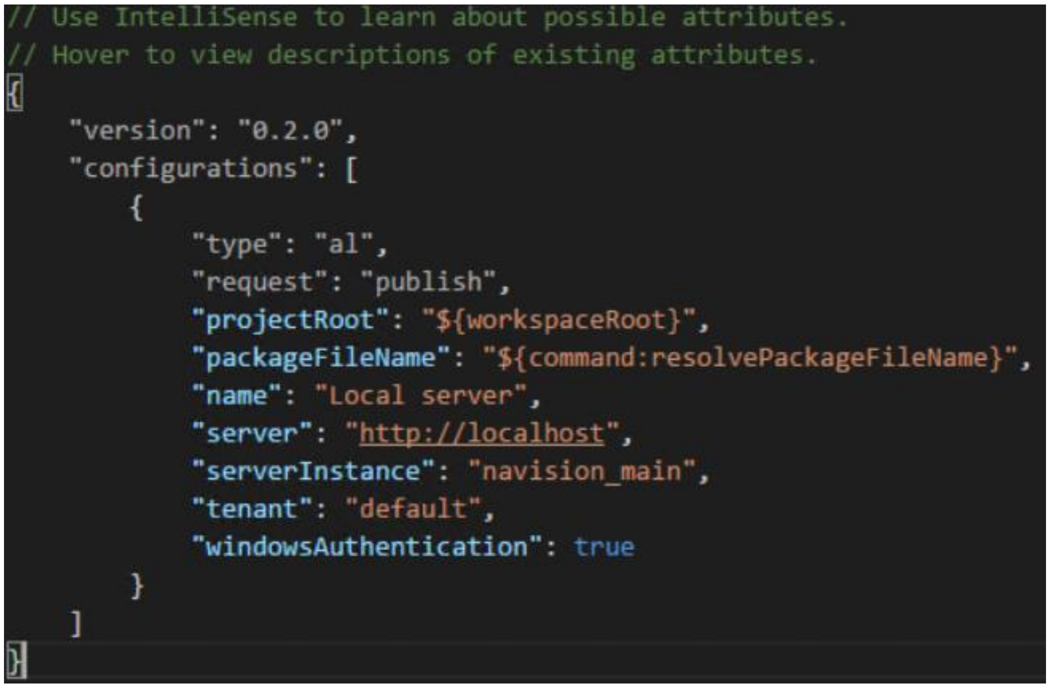
- Press CTRS+SHIFT+P
- Type "GO"
- Select the "AL:Go!" (You will get a question on where to put your files)
- Provide an empty folder, like "C:\Apps\ItemClassification"

3. Remove Unnecessary items

- Remove the "HelloWorld.al" file that is created automatically.

In the launch.json, remove the element "StartupObjectId" – we will not start any page after building the app for now. Also, in the same file, remove the configuration for "cloud".

Your Launch.json should look like:



```
// Use IntelliSense to learn about possible attributes.
// Hover to view descriptions of existing attributes.
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "al",
            "request": "publish",
            "projectRoot": "${workspaceRoot}",
            "packageFileName": "${command:resolvePackageFileName}",
            "name": "Local server",
            "server": "http://localhost",
            "serverInstance": "navision_main",
            "tenant": "default",
            "windowsAuthentication": true
        }
    ]
}
```

4. Add information to the app.json

The app.json is a kind of manifest file. Complete the following:

- Publisher: your last name
- Description: "This app adds the ability to classify your Items"

Results

A folder with the configured json-files ready to start developing your extension.

Exercise 2: Perform the necessary table changes

Exercise Scenario

The table changes that we need to do, are quite simple:

- A new table “Item Classification”

New field on the Item table

Task 1: Create Item Classification table

High Level Steps

1. Create a new file
2. Create a table with the snippet
3. Create the fields

Detailed Steps

1. Create a new file

- Click “New File” in VSCode Explorer
- Create the file “ItemClassificationTable.al

2. Create a table with the snippet

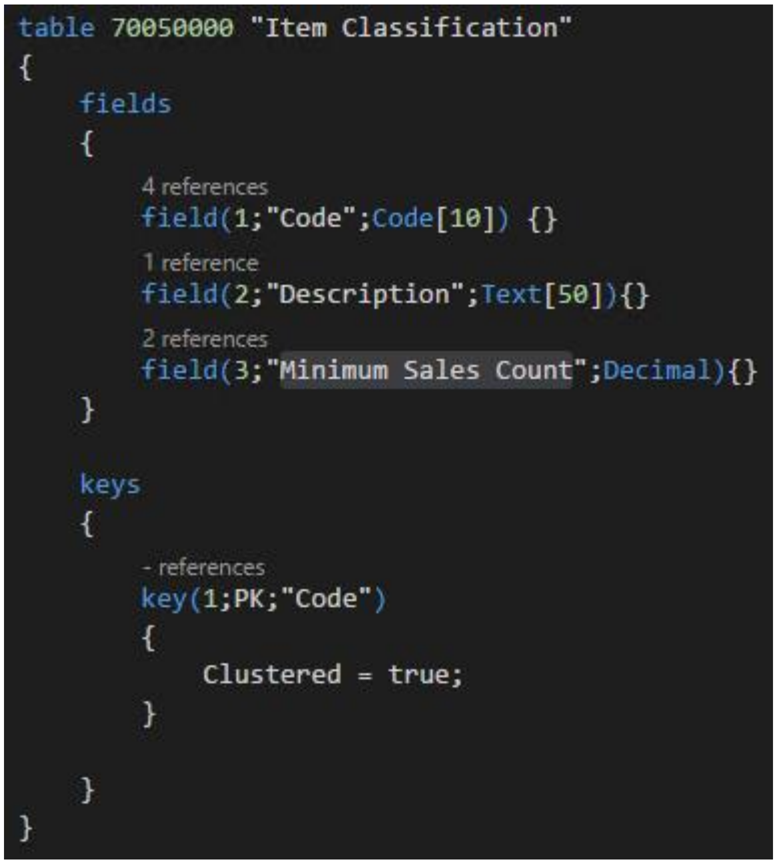
- Type “ttable” which will get you to the snippet of a table
- Change the ID to 70050000

Change “MyTable” to “Item Classification”

3. Create the fields

- Code (Code10)
- Description (Text50)
- Minimum Sales Count (Decimal)

The result should look something like this:


```
table 70050000 "Item Classification"
{
    fields
    {
        4 references
        field(1;"Code";Code[10]) {}
        1 reference
        field(2;"Description";Text[50]){}
        2 references
        field(3;"Minimum Sales Count";Decimal){}
    }

    keys
    {
        - references
        key(1;PK;"Code")
        {
            Clustered = true;
        }

    }
}
```

Task 2: Add the necessary field to the Item Table

High Level Steps

1. Add a Table Extension to extend the Item Table
2. Add the field

Detailed Steps

1. Add a Table Extension to extend the Item Table

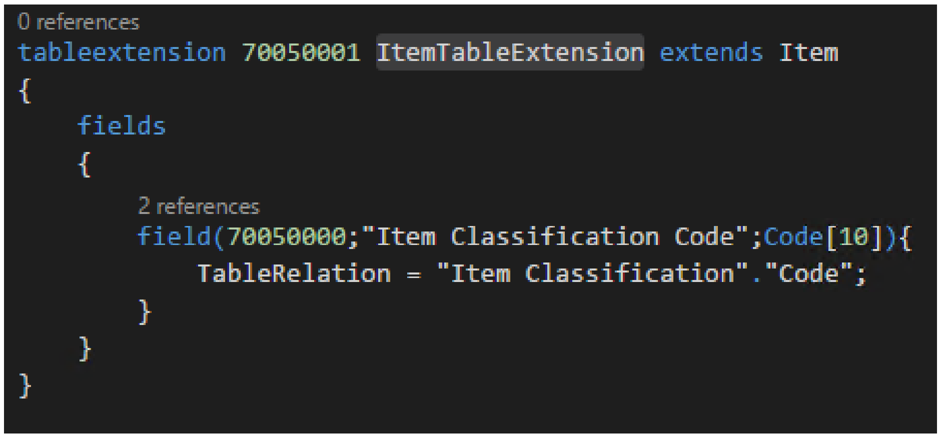
- Create a new file "ItemTableExtension.al"
- Use the snippet "ttableext" to get you started on the Table Extension.
- Change the ID to 70050001
- Change the name to "ItemTableExtension"

Make sure the TableExtension extends the "Item" table

2. Add the field

- Create a new field "Item Classification Code" (code 10)

The Table Extension should now look like this:



```
0 references
tableextension 70050001 ItemTableExtension extends Item
{
    fields
    {
        2 references
        field(70050000;"Item Classification Code";Code[10]){
            TableRelation = "Item Classification"."Code";
        }
    }
}
```

Task 3: Create the page for Item Classifications

High Level Steps

1. Create a new file
2. Use the snippets to create a new page

Detailed Steps

1. Create a new file

Create a new file “ItemClassificationPage” in the “Explorer” in VSCode

2. Use the snippets to create a new page

Type “tpage” and use the snippet

Change the following:

- ID to 70050000
- Name to “Item Classifications”
- PageType = List
- SourceTable = "Item Classification"
- Change “group” into “repeater” and give it the name “ItemClassifications”
- Add all the fields (tip: you can use snippet “tfieldpage”)
- Remove the entire “actions” part

Task 4: Test the Extension

High Level Steps

1. Change the launch.json to start your new page
2. Build the Extension and test in the web client

Detailed Steps

1. Change the launch.json to start your new page

- Open the launch.json (in the .vscode folder)

Add “"startupObjectId": 70050000”

2. Build the Extension and test in the web client

- Press F5

Exercise 3: Implement the “CalcItemClassification” method

Task 1: Create the Codeunit with the necessary business logic

High Level Steps

1. Create a new file
2. Create the codeunit with the snippet
3. Create the local function “GetItemSalesCount”
4. Create the local function “GetItemClassificationCode”
5. Now finish the Codeunit so that you can call the “CalcItemClassification” function

Detailed Steps

1. Create a new file

- We’ll create yet again a new file for the codeunit with the VSCode Explorer

Call it “CalcItemClassificationMethCodeunit.al”

2. Create the codeunit with the snippet

- Use snippet tcodeunit
- Change ID to 70050000
- Change name to “CalcItemClassification Meth.”
- Create a global procedure in the codeunit “CalcItemClassification” (tip: use snippet “tprocedure”)

Create a local procedure in that codeunit “DoCalcItemClassification”

3. Create the local function “GetItemSalesCount”

Add a new local function “GetItemSalesCount” with:

- Parameter “Item” (Record Item)
- Return value “Integer”
- Local var: ItemLedgerEntry (Record “Item Ledger Entry”)
- It returns the “count” of the number of “sales” in the Item Ledger

Entry for a certain item

Note: *Tip: you can copy the code from the previous Lab*

4. Create the local function "GetItemClassificationCode"

Add a new local function "GetItemClassificationCode" with:

- Parameter "SalesCount" (int)
- Return value "Code(10)"
- A variable "ItemClassification" (Rec)
- It should return the right classification code

Note: *Tip: you can copy the code from the previous Lab*

5. Now finish the Codeunit so that you can call the "CalcItemClassification" function

Task 2: Extend the codeunit with the necessary events to be able to decouple or extend the business logic.

High Level Steps

1. Add two Integration Events to the method
2. Implement the events with the "Handled VAR" pattern to be able to decouple or extend on this codeunit
3. Review the result

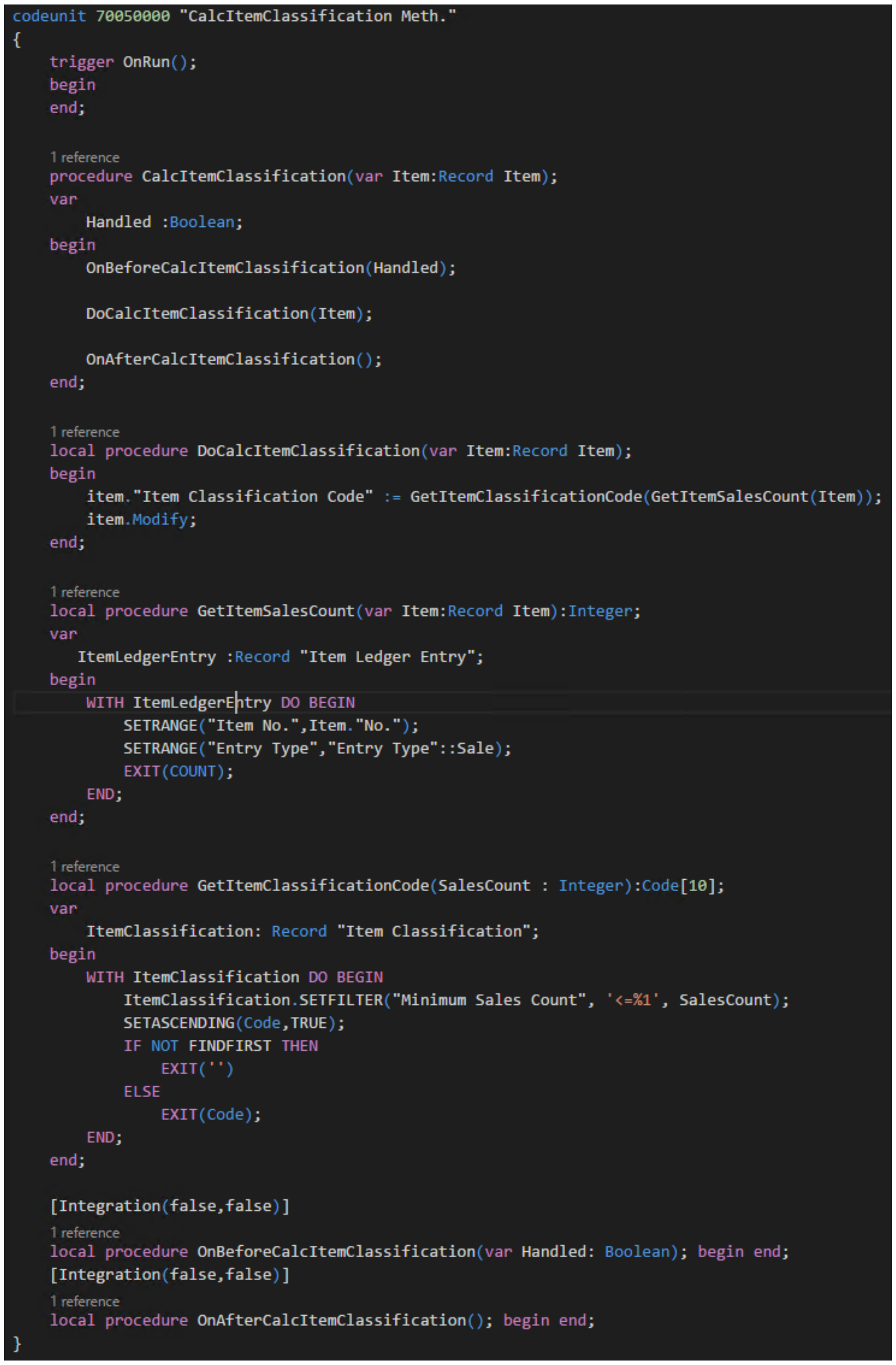
Detailed Steps

Add two Integration Events to the method

- OnBeforeCalcItemClassification
- OnAfterCalcItemClassification

Implement the events with the "Handled VAR" pattern to be able to decouple or extend on this codeunit

Review the result



```
codeunit 70050000 "CalcItemClassification Meth."
{
    trigger OnRun();
    begin
    end;

    1 reference
    procedure CalcItemClassification(var Item:Record Item);
    var
        Handled :Boolean;
    begin
        OnBeforeCalcItemClassification(Handled);

        DoCalcItemClassification(Item);

        OnAfterCalcItemClassification();
    end;

    1 reference
    local procedure DoCalcItemClassification(var Item:Record Item);
    begin
        item."Item Classification Code" := GetItemClassificationCode(GetItemSalesCount(Item));
        item.Modify;
    end;

    1 reference
    local procedure GetItemSalesCount(var Item:Record Item):Integer;
    var
       ItemLedgerEntry :Record "Item Ledger Entry";
    begin
        WITH ItemLedgerEntry DO BEGIN
            SETRANGE("Item No.",Item."No.");
            SETRANGE("Entry Type","Entry Type"::Sale);
            EXIT(COUNT);
        END;
    end;

    1 reference
    local procedure GetItemClassificationCode(SalesCount : Integer):Code[10];
    var
        ItemClassification: Record "Item Classification";
    begin
        WITH ItemClassification DO BEGIN
            ItemClassification.SETFILTER("Minimum Sales Count", '<=%1', SalesCount);
            SETASCENDING(Code,TRUE);
            IF NOT FINDFIRST THEN
                EXIT('')
            ELSE
                EXIT(Code);
        END;
    end;

    [Integration(false,false)]
    1 reference
    local procedure OnBeforeCalcItemClassification(var Handled: Boolean); begin end;
    [Integration(false,false)]
    1 reference
    local procedure OnAfterCalcItemClassification(); begin end;
}
```

Task 3: On the Item list, we need to be able to call the method. And we need to see the field there as well.

High Level Steps

1. Create a page extension for the “Item List”
2. Add the Action “Calculate Item Classification”

3. Add the field on the same Item List
4. Review the result

Detailed Steps

1. Create a page extension for the "Item List"

- Create the file "ItemListPageExtension.al" in the VSCode Explorer
- Use the snippet "tpageext" to create a page extension
- Change the ID to 70050001
- Name it "ItemListPageExtension"

Make use it extends the Item List

2. Add the Action "Calculate Item Classification"

- In the Actions-part, provide an element "AddFirst" (we want to add our action in the beginning" of the "Item" group. (use intellisense)
- Add an action and put code in the action, where you call the "CalcItemClassification" method. For this, create a "OnAction" trigger (use the ttrigger snippet).

Set these properties:

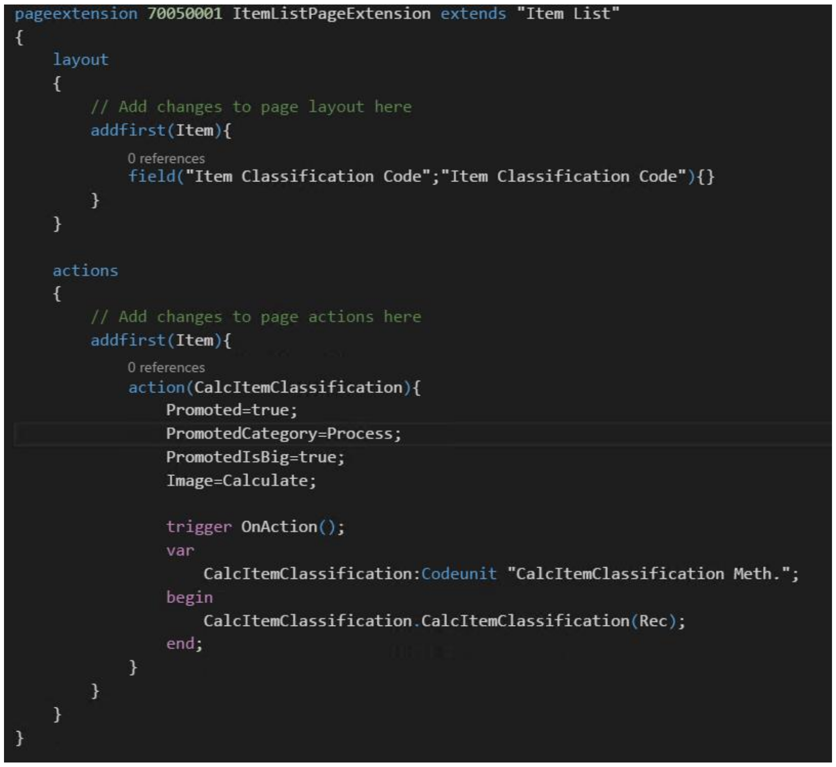
- Promoted=true;
- PromotedCategory=Process;
- PromotedIsBig=true;

Image=Calculate;

3. Add the field on the same Item List

- Add a field in the "layout"-section of the same page-extension.
- Also use an "addfirst" element to the "Item" group. In this element, add a field "Item Classification Code".

4. Review the result


```
pageextension 70050001 ItemListPageExtension extends "Item List"
{
    layout
    {
        // Add changes to page layout here
        addfirst(Item){
            0 references
            field("Item Classification Code";"Item Classification Code"){}
        }
    }

    actions
    {
        // Add changes to page actions here
        addfirst(Item){
            0 references
            action(CalcItemClassification){
                Promoted=true;
                PromotedCategory=Process;
                PromotedIsBig=true;
                Image=Calculate;

                trigger OnAction();
                var
                    CalcItemClassification:Codeunit "CalcItemClassification Meth.";
                begin
                    CalcItemClassification.CalcItemClassification(Rec);
                end;
            }
        }
    }
}
```

Task 4: Test the functionality

High Level Steps

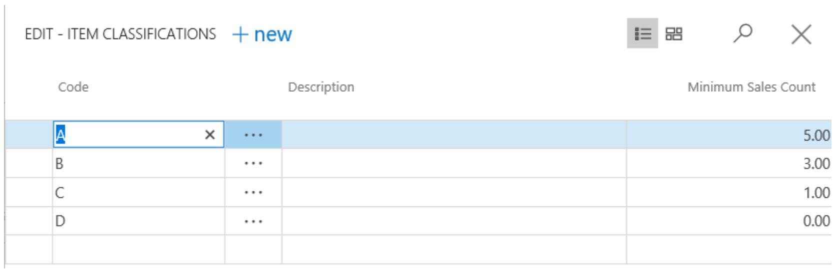
1. Build the App
2. Add some data in the "Item Classification" table
3. Navigate to the Item List page

Detailed Steps

1. Build the App

Press F5

2. Add some data in the "Item Classification" table



Code	Description	Minimum Sales Count
A		5.00
B		3.00
C		1.00
D		0.00

3. Navigate to the Item List page

Find your new action and execute it for a few items.

Exercise 4: Implement notifications

Task 1: Create an extra field in the “Item Classification” table

High Level Steps

1. Add the field
2. Put created field on the “Item Classifications” page

Detailed Steps

1. Add the field
- Open the file “ItemClassificationTable.al”
- Add the field “Warning”, which is a Boolean.
2. Put created field on the “Item Classifications” page
- Open the file “ItemClassificationPage.al”
- Put that field on the repeater, as a last field.

Task 2: Create the Codeunit with the necessary business logic

High Level Steps

1. Create a typical Method Codeunit
2. Create the local function “GetItemClassWarningStatus”
3. Create the public function “HandleItemClassWarning_RunItemCard”
4. Create the public function “HandleItemClassWarning_RunItemLedgerEntries”
5. Create the local function “SendItemClassWarning”
6. Create the Event Subscriber “ShowItemClassWarning_OnValidateItemNoOnSalesLine”
7. Now finish the Codeunit so that you can call the “ShowItemClassWarning” function

Detailed Steps

1. Create a typical Method Codeunit
- Create a new file “ShowItemClassificationWarningMethCodeunit.al”
- Create a new Codeunit “ShowItemClassWarning Method” with the snippet (ID 70050001)
- Create a global function in the Codeunit “ShowItemClassWarning”

Create a local function in that Codeunit “DoShowItemClassWarning”

- Create a new file “ShowItemClassificationWarningMethCodeunit.al”
- Create a new Codeunit “ShowItemClassWarning Method” with the snippet (ID 70050001)

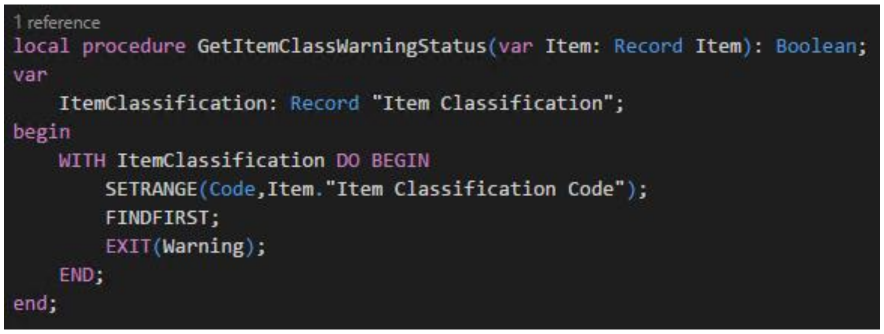
- Create a global function in the Codeunit “ShowItemClassWarning”

Create a local function in that Codeunit “DoShowItemClassWarning”

2. Create the local function “GetItemClassWarningStatus”

- The function should return a true or false, whether an Item’s classification should show a warning or not
- Parameter: Item (Record)
- Return type: Boolean
- Variable “ItemClassification” (Record)

It should look like this:



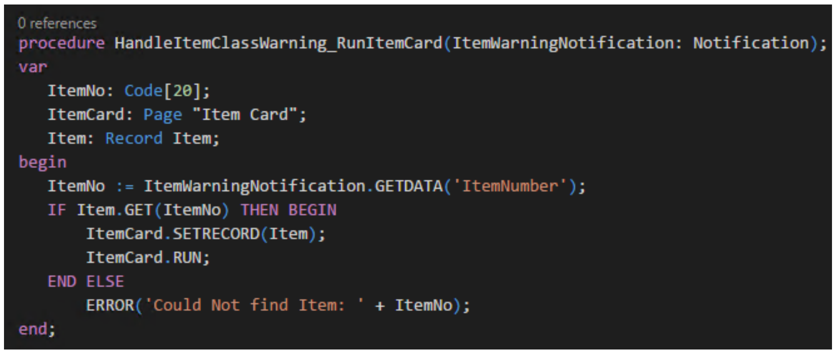
```
1 reference
local procedure GetItemClassWarningStatus(var Item: Record Item): Boolean;
var
    ItemClassification: Record "Item Classification";
begin
    WITH ItemClassification DO BEGIN
        SETRANGE(Code,Item."Item Classification Code");
        FINDFIRST;
        EXIT(Warning);
    END;
end;
```

3. Create the public function “HandleItemClassWarning_RunItemCard”

This is going to be the “Handler” of the notification (When the user wants to see the “Item Card”).

- Parameter: “ItemWarningNotification” (Notification)
- The business logic should get the ItemNumber from the Notification, and should raise an error when the Item Number does not exist.

This should be the result:

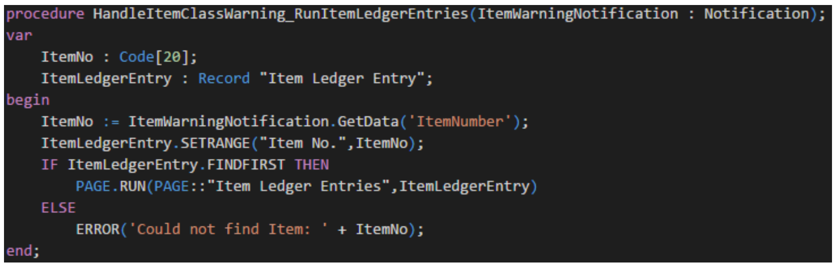


```
0 references
procedure HandleItemClassWarning_RunItemCard(ItemWarningNotification: Notification);
var
   ItemNo: Code[20];
   ItemCard: Page "Item Card";
   Item: Record Item;
begin
   ItemNo := ItemWarningNotification.GETDATA('ItemNumber');
   IF Item.GET(ItemNo) THEN BEGIN
       ItemCard.SETRECORD(Item);
       ItemCard.RUN;
   END ELSE
       ERROR('Could Not find Item: ' + ItemNo);
end;
```

4. Create the public function “HandleItemClassWarning_RunItemLedgerEntries”

Let’s assume we also want the user to be able to drill into the Item Ledger Entries. We need a handler for this as well, very similar to the previous one.

This could be the result:



```
procedure HandleItemClassWarning_RunItemLedgerEntries(ItemWarningNotification : Notification);
var
    ItemNo : Code[20];
    ItemLedgerEntry : Record "Item Ledger Entry";
begin
    ItemNo := ItemWarningNotification.GetData('ItemNumber');
    ItemLedgerEntry.SETRANGE("Item No.",ItemNo);
    IF ItemLedgerEntry.FINDFIRST THEN
        PAGE.RUN(PAGE::"Item Ledger Entries",ItemLedgerEntry)
    ELSE
        ERROR('Could not find Item: ' + ItemNo);
end;
```

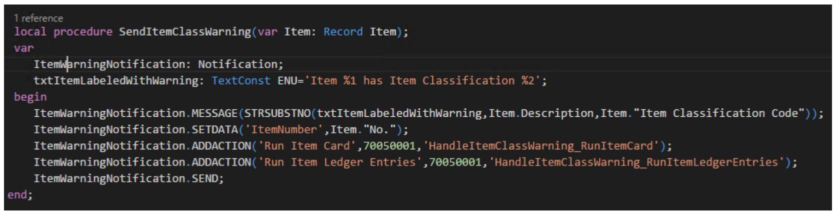
5. Create the local function “SendItemClassWarning”

This is where we will send the notification.

These are the characteristics:

- Parameter: Item (Record)
- Variable “ItemWarningNotification” (Notification)
- Text Constant: “txtItemLabeledWithWarning” (Item %1 has Item Classification %2). Add your own language as well.
- The function should raise the notification, and add the two previously defined actions.

This should be the result:



```
1 reference
local procedure SendItemClassWarning(var Item: Record Item);
var
    ItemWarningNotification: Notification;
    txtItemLabeledWithWarning: TextConst ENU='Item %1 has Item Classification %2';
begin
    ItemWarningNotification.MESSAGE(STRSUBSTNO(txtItemLabeledWithWarning,Item.Description,Item."Item Classification Code"));
    ItemWarningNotification.SETDATA('ItemNumber',Item."No.");
    ItemWarningNotification.ADDACTION('Run Item Card',70050001,'HandleItemClassWarning_RunItemCard');
    ItemWarningNotification.ADDACTION('Run Item Ledger Entries',70050001,'HandleItemClassWarning_RunItemLedgerEntries');
    ItemWarningNotification.SEND;
end;
```

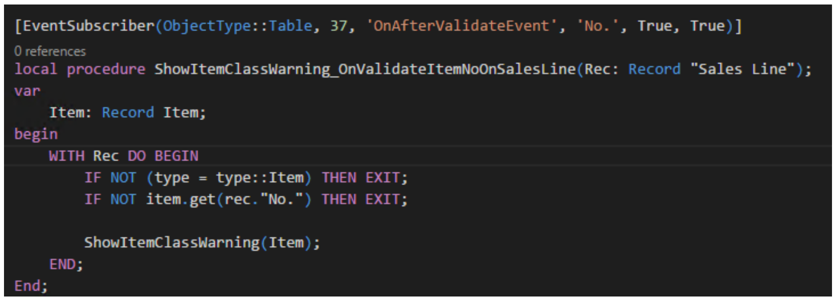
6. Create the Event Subscriber “ShowItemClassWarning_OnValidateItemNoOnSalesLine”

Now, we need to define where we will call this notification. In this case, we want to send the notification on the Sales Line, when validating an Item No.

- Create a subscriber to the OnAfterValidateEvent of Table 37 and field “No.”.

- Parameter: Rec (Sales Line)
- Variable: Item (Record)

The business logic should be this:



```
[EventSubscriber(ObjectType::Table, 37, 'OnAfterValidateEvent', 'No.', True, True)]
0 references
local procedure ShowItemClassWarning_OnValidateItemNoOnSalesLine(Rec: Record "Sales Line");
var
    Item: Record Item;
begin
    WITH Rec DO BEGIN
        IF NOT (type = type::Item) THEN EXIT;
        IF NOT item.get(rec."No.") THEN EXIT;

        ShowItemClassWarning(Item);
    END;
End;
```

7. Now finish the Codeunit so that you can call the "ShowItemClassWarning" function

Add the Item-record as a parameter to functions:

- ShowItemClassWarning
- DoShowItemClassWarning

Complete the latter with this business logic:

IF GetItemClassWarningStatus(Item) THEN
SendItemClassWarning(Item);

High Level Steps

Extend the Codeunit with the necessary events to be able to decouple or extend the business logic

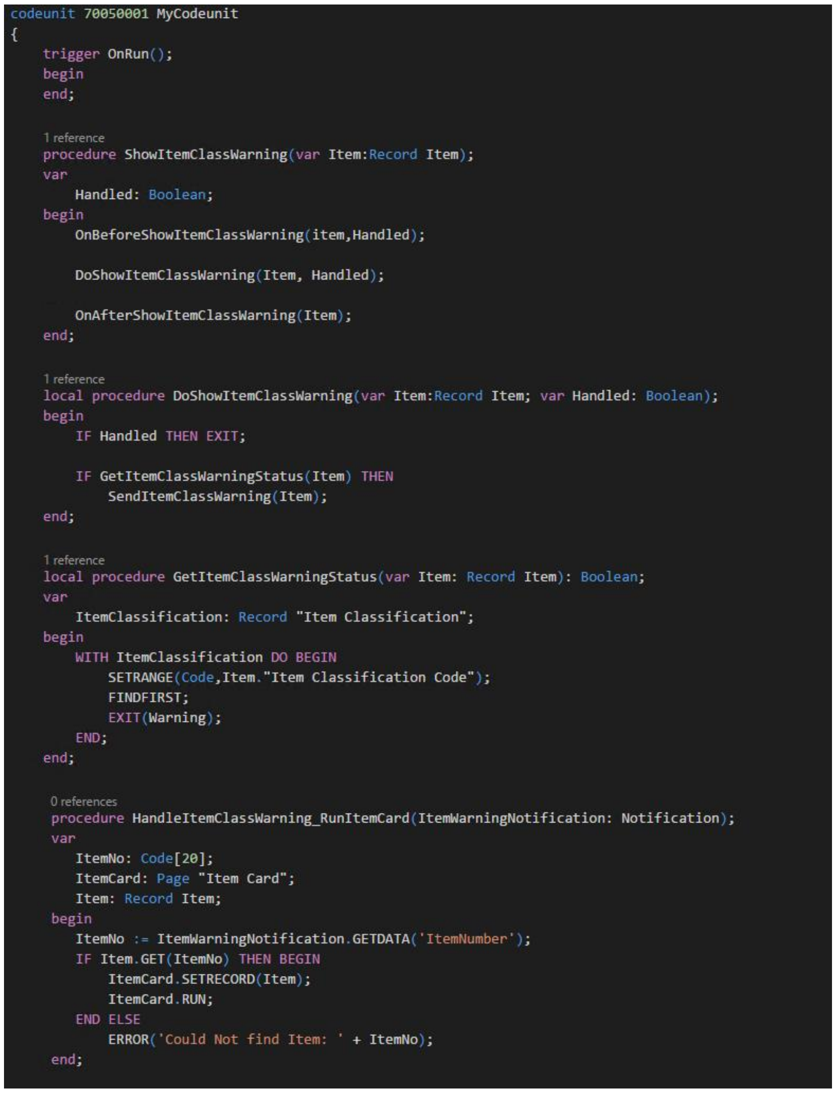
Detailed Steps

Extend the Codeunit with the necessary events to be able to decouple or extend the business logic

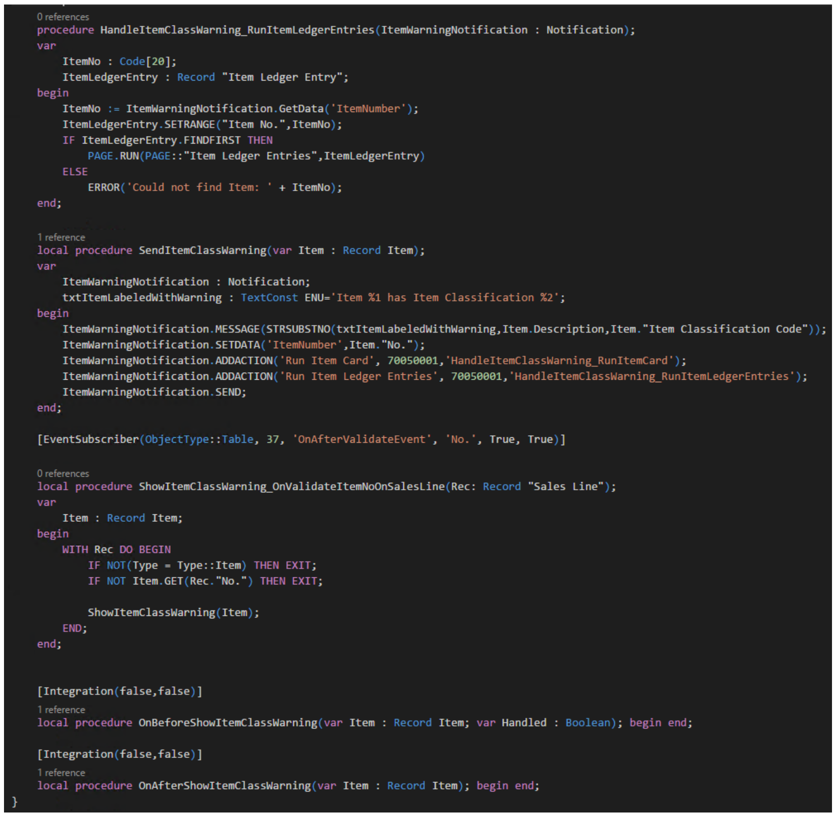
Just like the previous method, implement the events:

- OnBeforeShowItemClassWarning
- OnAfterShowItemClassWarning

The finished result of the codeunit should be…



```
codeunit 70050001 MyCodeunit
{
    trigger OnRun();
    begin
    end;

    1 reference
    procedure ShowItemClassWarning(var Item:Record Item);
    var
        Handled: Boolean;
    begin
        OnBeforeShowItemClassWarning(item,Handled);

        DoShowItemClassWarning(Item, Handled);

        OnAfterShowItemClassWarning(Item);
    end;

    1 reference
    local procedure DoShowItemClassWarning(var Item:Record Item; var Handled: Boolean);
    begin
        IF Handled THEN EXIT;

        IF GetItemClassWarningStatus(Item) THEN
            SendItemClassWarning(Item);
    end;

    1 reference
    local procedure GetItemClassWarningStatus(var Item: Record Item): Boolean;
    var
        ItemClassification: Record "Item Classification";
    begin
        WITH ItemClassification DO BEGIN
            SETRANGE(Code,Item."Item Classification Code");
            FINDFIRST;
            EXIT(Warning);
        END;
    end;

     0 references
     procedure HandleItemClassWarning_RunItemCard(ItemWarningNotification: Notification);
     var
        ItemNo: Code[20];
        ItemCard: Page "Item Card";
        Item: Record Item;
     begin
        ItemNo := ItemWarningNotification.GETDATA('ItemNumber');
        IF Item.GET(ItemNo) THEN BEGIN
            ItemCard.SETRECORD(Item);
            ItemCard.RUN;
        END ELSE
            ERROR('Could Not find Item: ' + ItemNo);
     end;
```



```
    0 references
    procedure HandleItemClassWarning_RunItemLedgerEntries(ItemWarningNotification : Notification);
    var
        ItemNo : Code[20];
        ItemLedgerEntry : Record "Item Ledger Entry";
    begin
        ItemNo := ItemWarningNotification.GetData('ItemNumber');
        ItemLedgerEntry.SETRANGE("Item No.",ItemNo);
        IF ItemLedgerEntry.FINDFIRST THEN
            PAGE.RUN(PAGE::"Item Ledger Entries",ItemLedgerEntry)
        ELSE
            ERROR('Could not find Item: ' + ItemNo);
    end;

    1 reference
    local procedure SendItemClassWarning(var Item : Record Item);
    var
        ItemWarningNotification : Notification;
        txtItemLabeledWithWarning : TextConst ENU='Item %1 has Item Classification %2';
    begin
        ItemWarningNotification.MESSAGE(STRSUBSTNO(txtItemLabeledWithWarning,Item.Description,Item."Item Classification Code"));
        ItemWarningNotification.SETDATA('ItemNumber',Item."No.");
        ItemWarningNotification.ADDACTION('Run Item Card', 70050001,'HandleItemClassWarning_RunItemCard');
        ItemWarningNotification.ADDACTION('Run Item Ledger Entries', 70050001,'HandleItemClassWarning_RunItemLedgerEntries');
        ItemWarningNotification.SEND;
    end;

    [EventSubscriber(ObjectType::Table, 37, 'OnAfterValidateEvent', 'No.', True, True)]

    0 references
    local procedure ShowItemClassWarning_OnValidateItemNoOnSalesLine(Rec: Record "Sales Line");
    var
        Item : Record Item;
    begin
        WITH Rec DO BEGIN
            IF NOT(Type = Type::Item) THEN EXIT;
            IF NOT Item.GET(Rec."No.") THEN EXIT;

            ShowItemClassWarning(Item);
        END;
    end;


    [Integration(false,false)]
    1 reference
    local procedure OnBeforeShowItemClassWarning(var Item : Record Item; var Handled : Boolean); begin end;

    [Integration(false,false)]
    1 reference
    local procedure OnAfterShowItemClassWarning(var Item : Record Item); begin end;
}
```

Task 4: Test the functionality

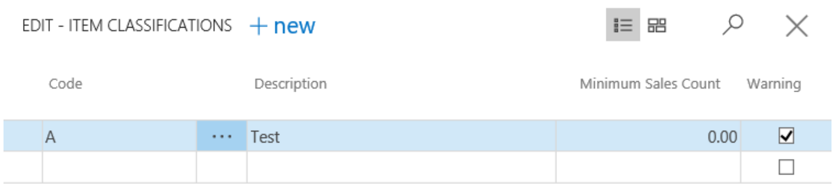
High Level Steps

Build the Extension and Test in the Web Client

Detailed Steps

Build the Extension and Test in the Web Client

- Press F5
- Enter Data like this:



Code	Description	Minimum Sales Count	Warning
A	Test	0.00	☑
			☐

FIGURE .46:ONLY ONE RECORD WITH MINIMUM TO 0, SO THAT IT WILL ALWAYS SHOW A WARNING.

- Calculate the classification of an item
- Use that Item in a Sales line

When the notification is show, try both available actions ("Run Item Card" and "Run Item Ledger Entries")

Exercise 5: Wizard for initial Setup

Exercise Scenario

We will create a Wizard page for the users to make it easier for them to initially create configuration data for the Extension.

The idea is again quite simple: if there is no data, the Wizard will notify that there is none, and foresee data for the user as "default data", which the user can change for himself.

If there IS data, there is no need for the Wizard.

Task 1: Create a new page

High Level Steps

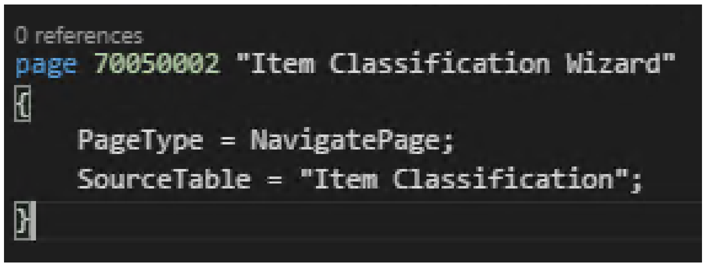
Create a new page

Detailed Steps

Create a new page

- Create a new file with the name "ItemClassificationWizardPage.al"
- Create a new page (not using a snippet)
  - ID: 70050002
  - Name: Item Classification Wizard
  - PageType: NavigatePage
  - SourceTable: Item Classification

Result:



```
0 references
page 70050002 "Item Classification Wizard"
{
    PageType = NavigatePage;
    SourceTable = "Item Classification";
}
```

Task 2: Add steps to a wizard page

High Level Steps

1. Add two groups to the page.

2. Create global variables that will be used for setting Group visibility
3. Define Group visibility
4. Implement the first step
5. Implement the second step
6. Review the result

Detailed Steps

1. Add two groups to the page.

- Add a "Layout" section
- Within this layout, create an area (content)
- Within this area, we can create the two groups. Each group will represent one step in the Wizard:
    - FirstStep
    - SecondStep.

2. Create global variables that will be used for setting Group visibility

- At the end of the page, create the "var" section and create two variables:
- FirstStepVisible (Boolean)
- SecondStepVisible (Boolean)

3. Define Group visibility in both groups:

- add the properties "Visible"
- change visible to the newly created variable

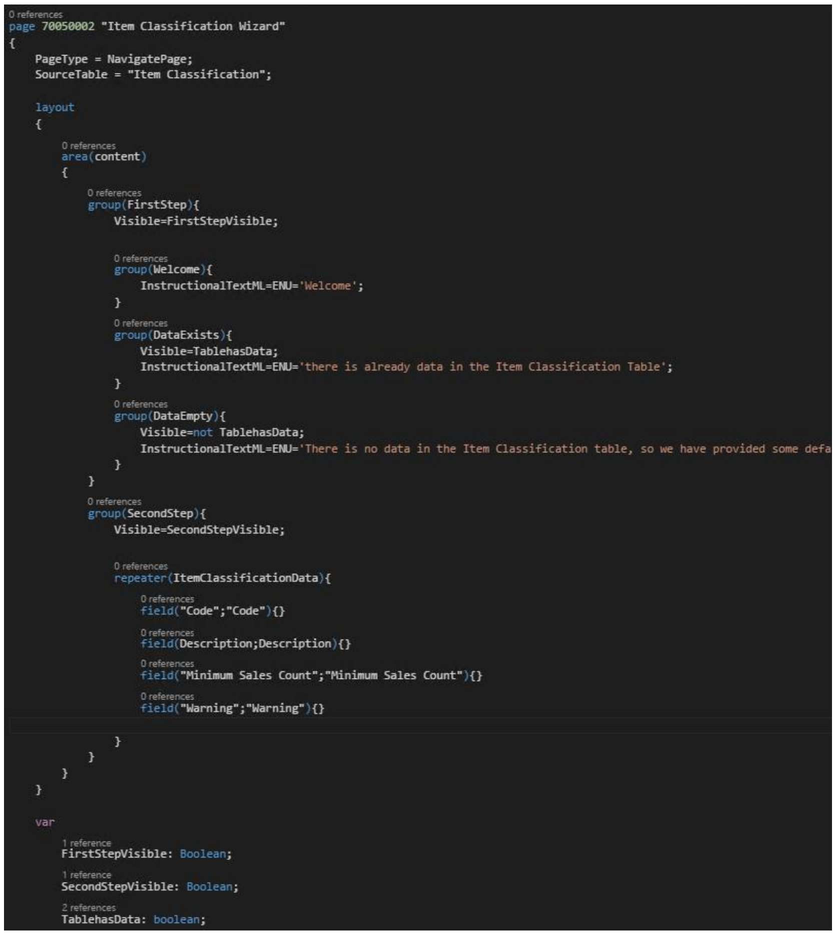
4. Implement the first step

- Add the global var "TableHasData" (Boolean)
- Add three subgroups to the first step.
    - Group(Welcome)
    - Group(DataExists)
    - Group(DataEmpty)
- Set properties of first subgroup:
    - InstructionalTextML: "Welcome"
- Set properties of second subgroup:
    - Visible: TableHasData
    - InstructionalTextML: "There is already data in the Item Classification table."
- Set properties of third subgroup:
    - Visible: NOT TableHasData
    - InstructionalTextML: "There is no data in the Item Classification table, so we have provided some default values. Click Next and review the data."

5. Implement the second step

- Add subgroup to the group “SecondStep” and name it “ItemClassificationData” and make it a repeater.
- Add all fields to the repeater

6. Review the result



```
0 references
page 70050002 "Item Classification Wizard"
{
    PageType = NavigatePage;
    SourceTable = "Item Classification";

    layout
    {
        0 references
        area(content)
        {
            0 references
            group(FirstStep){
                Visible=FirstStepVisible;


                0 references
                group(Welcome){
                    InstructionalTextML=ENU='Welcome';
                }
                0 references
                group(DataExists){
                    Visible=TablehasData;
                    InstructionalTextML=ENU='there is already data in the Item Classification Table';
                }
                0 references
                group(DataEmpty){
                    Visible=not TablehasData;
                    InstructionalTextML=ENU='There is no data in the Item Classification table, so we have provided some defa
                }
            }
            0 references
            group(SecondStep){
                Visible=SecondStepVisible;


                0 references
                repeater(ItemClassificationData){
                    0 references
                    field("Code";"Code"){}
                    0 references
                    field(Description;Description){}
                    0 references
                    field("Minimum Sales Count";"Minimum Sales Count"){}
                    0 references
                    field("Warning";"Warning"){}

                }
            }
        }
    }

    var
        1 reference
        FirstStepVisible: Boolean;
        1 reference
        SecondStepVisible: Boolean;
        2 references
        TablehasData: boolean;
```

Task 3: Implement the “InsertDefaultValue” logic for this Wizard

High Level Steps

Create a new method codeunit that inserts three records in the “Item Classification” table.

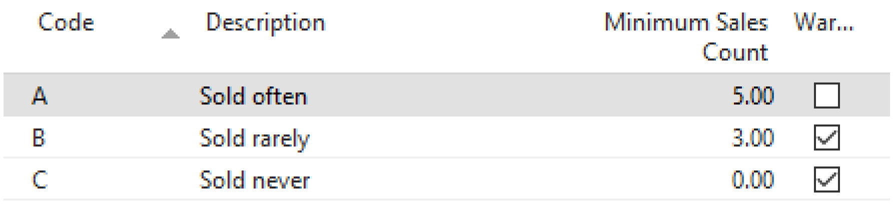
Call this codeunit from a method on the table

Detailed Steps

Create a new method codeunit that inserts three records in the “Item Classification” table.

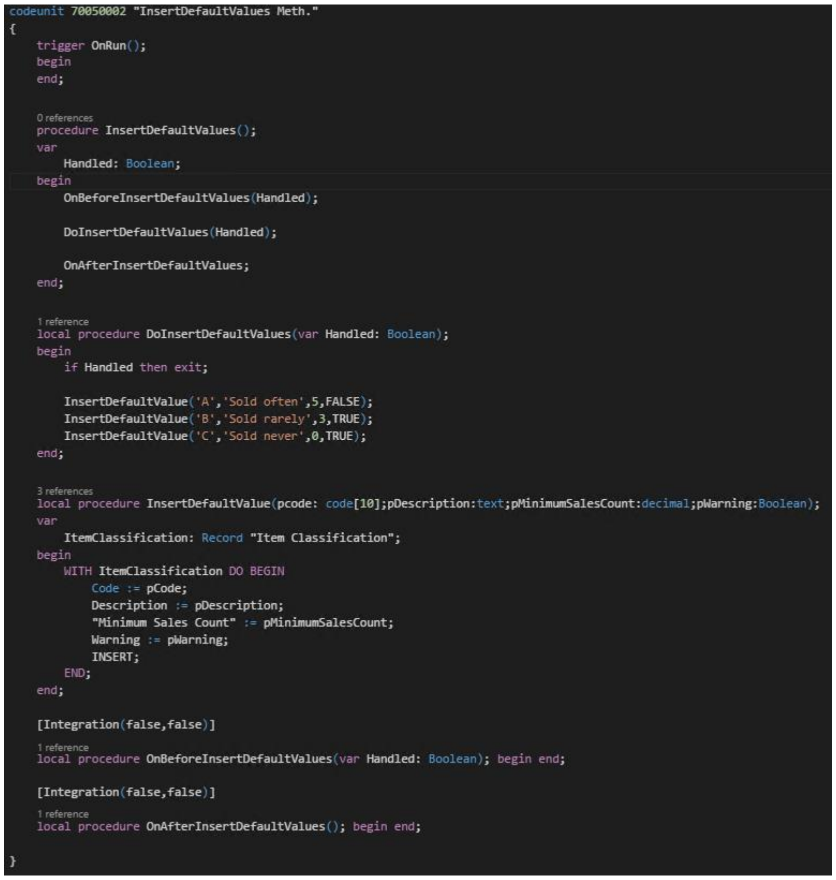
- Create a new file “InsertDefaultValuesMethCodeunit.al”

- Create a typical method-codeunit with the name “InsertDefaultValues Meth.”
- Implement the extensibility with OnBefore- and OnAfter-events
- Create a function that inserts one specific record
- Call that function three times to create these three records



Code	Description	Minimum Sales Count	War...
A	Sold often	5.00	☐
B	Sold rarely	3.00	✓
C	Sold never	0.00	✓

And this should be your codeunit:

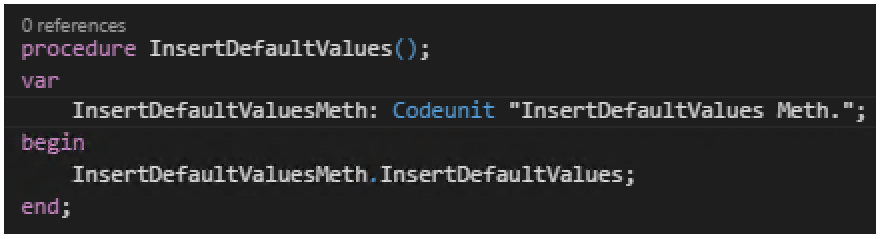


```
codeunit 70050002 "InsertDefaultValues Meth."
{
    trigger OnRun();
    begin
    end;

    0 references
    procedure InsertDefaultValues();
    var
        Handled: Boolean;
    begin
        OnBeforeInsertDefaultValues(Handled);

        DoInsertDefaultValues(Handled);

        OnAfterInsertDefaultValues;
    end;

    1 reference
    local procedure DoInsertDefaultValues(var Handled: Boolean);
    begin
        if Handled then exit;

        InsertDefaultValue('A','Sold often',5,FALSE);
        InsertDefaultValue('B','Sold rarely',3,TRUE);
        InsertDefaultValue('C','Sold never',0,TRUE);
    end;

    3 references
    local procedure InsertDefaultValue(pcode: code[10];pDescription:text;pMinimumSalesCount:decimal;pWarning:Boolean);
    var
        ItemClassification: Record "Item Classification";
    begin
        WITH ItemClassification DO BEGIN
            Code := pCode;
            Description := pDescription;
            "Minimum Sales Count" := pMinimumSalesCount;
            Warning := pWarning;
            INSERT;
        END;
    end;

    [Integration(false,false)]
    1 reference
    local procedure OnBeforeInsertDefaultValues(var Handled: Boolean); begin end;

    [Integration(false,false)]
    1 reference
    local procedure OnAfterInsertDefaultValues(); begin end;

}
```

Call this codeunit from a method on the table

- Create a new function “InsertDefaultValues” in table “Item Classification”

- Call the global function in the new codeunit



```
0 references
procedure InsertDefaultValues();
var
    InsertDefaultValuesMeth: Codeunit "InsertDefaultValues Meth.";
begin
    InsertDefaultValuesMeth.InsertDefaultValues;
end;
```

Task 4: Add Navigation to the Wizard

High Level Steps

1. Create Navigation Actions on the Item Classification Wizard Page
2. Create the local function “ResetControls”
3. Create the local function “EnableControls”
4. Create the local function “ShowFirstStep”
5. Create the local function “ShowSecondStep”
6. Create the local function “NextStep”
7. Create the local function “FinishAction”
8. Create navigation logic

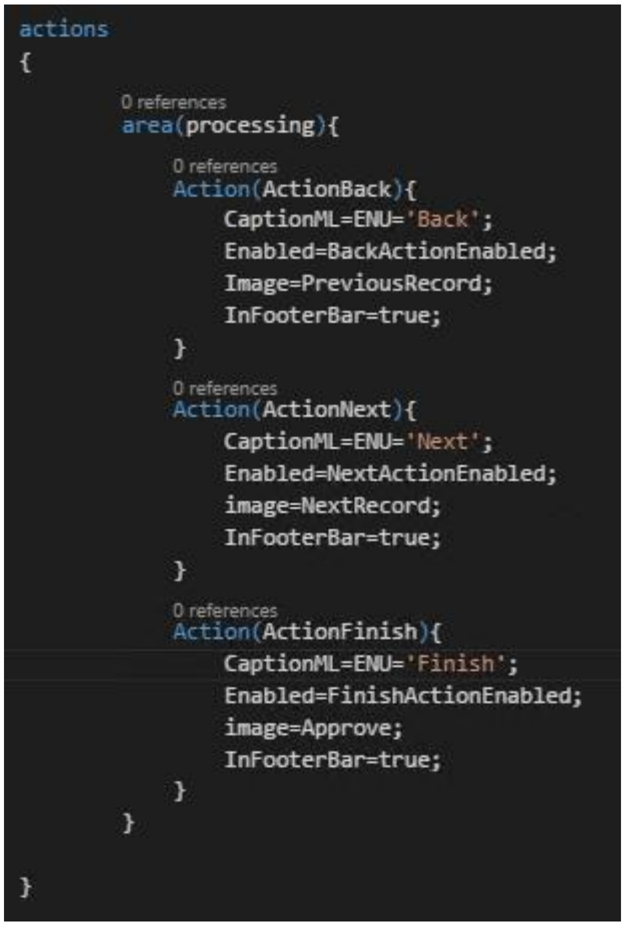
Detailed Steps

1. Create Navigation Actions on the Item Classification Wizard Page

- Open file “ItemClassificationWizardPage.al”
- Add an Actions section after the “layout” section
- In this section, create an area (with the name “processing”)
- Now at three Actions:
    - "ActionBack” with Caption: “Back”
    - "ActionNext” with Caption: “Next”
    - "ActionFinish” with Caption: “Finish”
- Create global variables that will be used for enabling navigation
    - Step (Option: ”First,Second”)
    - BackActionEnabled (Boolean)
    - NextActionEnabled (Boolean)
    - FinishActionEnabled (Boolean)
- Properties for the “ActionBack” are:
    - “Enabled”: “BackActionEnabled”
    - “Image”: “PreviousRecord”
    - “InFooterBar”: true
- Properties for the “ActionNext” are:

    - “Enabled”: “NextActionEnabled”
    - “Image”: “NextRecord”
    - “InFooterBar”: true
- Properties for the “ActionFinish” are:
    - “Enabled”: “FinishActionEnabled”
    - “Image”: “Approve”
    - “InFooterBar”: true

The Actions section should now look like this:

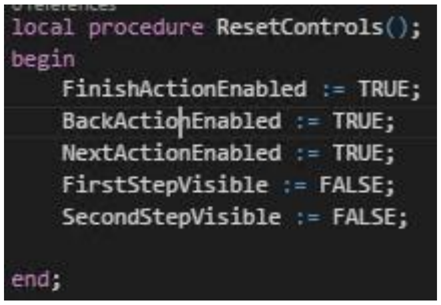


```
actions
{
        0 references
        area(processing){
            0 references
            Action(ActionBack){
                CaptionML=ENU='Back';
                Enabled=BackActionEnabled;
                Image=PreviousRecord;
                InFooterBar=true;
            }
            0 references
            Action(ActionNext){
                CaptionML=ENU='Next';
                Enabled=NextActionEnabled;
                image=NextRecord;
                InFooterBar=true;
            }
            0 references
            Action(ActionFinish){
                CaptionML=ENU='Finish';
                Enabled=FinishActionEnabled;
                image=Approve;
                InFooterBar=true;
            }
        }

}
```

o

2. Create the local function “ResetControls”

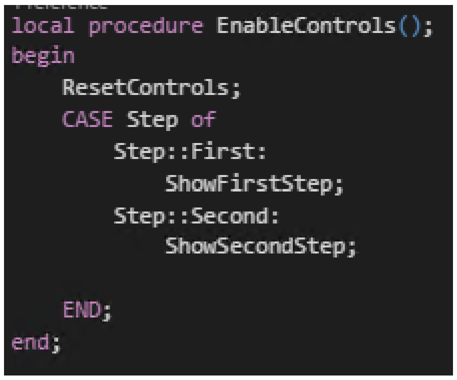
This function is going to reset all setup for the controls we have.



```
local procedure ResetControls();
begin
    FinishActionEnabled := TRUE;
    BackActionEnabled := TRUE;
    NextActionEnabled := TRUE;
    FirstStepVisible := FALSE;
    SecondStepVisible := FALSE;

end;
```

3. Create the local function “EnableControls”

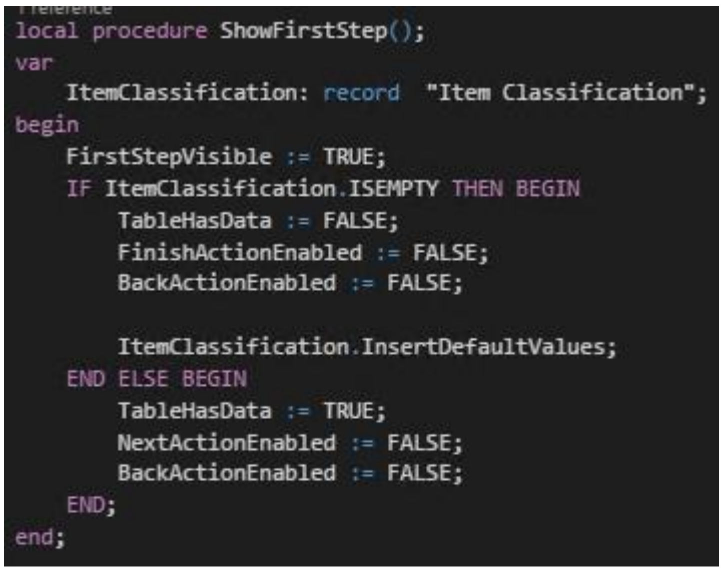
This function will enable the right controls depending on the step we’re in:



```
local procedure EnableControls();
begin
    ResetControls;
    CASE Step of
        Step::First:
            ShowFirstStep;
        Step::Second:
            ShowSecondStep;

    END;
end;
```

3. Create the local function “ShowFirstStep”

This could be the logic.

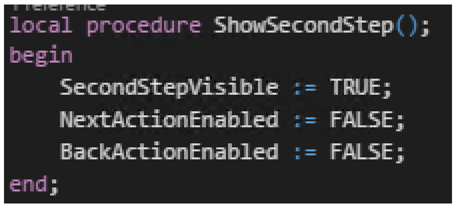


```
local procedure ShowFirstStep();
var
    ItemClassification: record  "Item Classification";
begin
    FirstStepVisible := TRUE;
    IF ItemClassification.ISEMPTY THEN BEGIN
        TableHasData := FALSE;
        FinishActionEnabled := FALSE;
        BackActionEnabled := FALSE;

        ItemClassification.InsertDefaultValues;
    END ELSE BEGIN
        TableHasData := TRUE;
        NextActionEnabled := FALSE;
        BackActionEnabled := FALSE;
    END;
end;
```

You could debate whether it is a good thing to have our records created at this point. Basically it can – as part of the Wizard and the Wizard only.

4. Create the local function "ShowSecondStep"

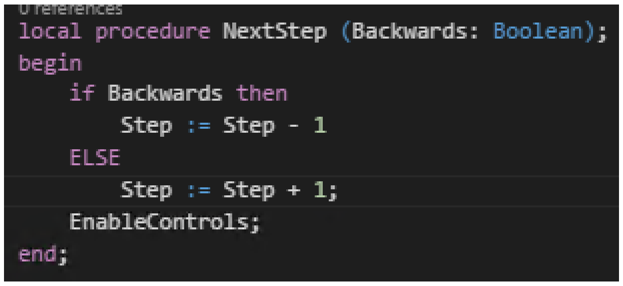
And this is what the second step looks like (showing the table):



```
1 reference
local procedure ShowSecondStep();
begin
    SecondStepVisible := TRUE;
    NextActionEnabled := FALSE;
    BackActionEnabled := FALSE;
end;
```

5. Create the local function "NextStep"

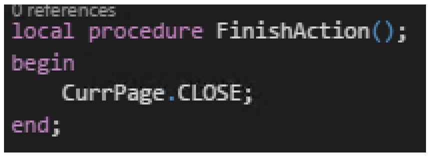
We also need to handle the steps:



```
0 references
local procedure NextStep (Backwards: Boolean);
begin
    if Backwards then
        Step := Step - 1
    ELSE
        Step := Step + 1;
    EnableControls;
end;
```

6. Create the local function "FinishAction"

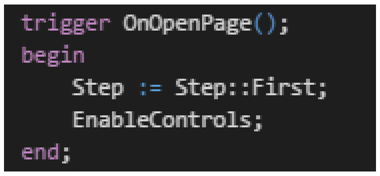
And we keep the "Finish" quite simple:



```
0 references
local procedure FinishAction();
begin
    CurrPage.CLOSE;
end;
```

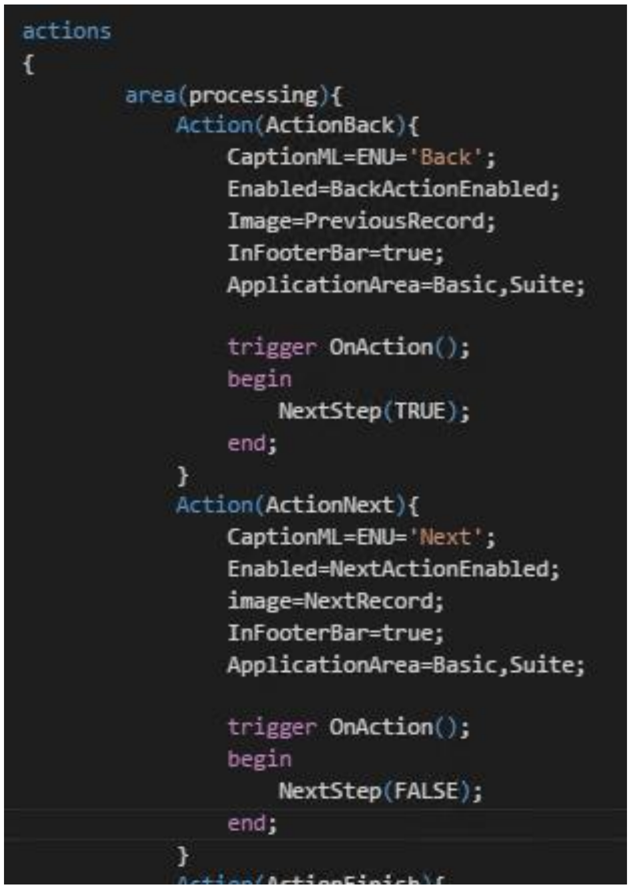
7. Create navigation logic

- On the "OnOpenPage" trigger, add this piece of code:



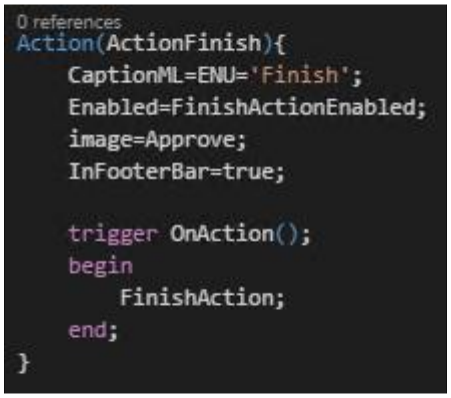
```
trigger OnOpenPage();
begin
    Step := Step::First;
    EnableControls;
end;
```

- On "ActionBack" and "ActionNext", we need to call the "NextStep" function



```
actions
{
        area(processing){
            Action(ActionBack){
                CaptionML=ENU='Back';
                Enabled=BackActionEnabled;
                Image=PreviousRecord;
                InFooterBar=true;
                ApplicationArea=Basic,Suite;

                trigger OnAction();
                begin
                    NextStep(TRUE);
                end;
            }
            Action(ActionNext){
                CaptionML=ENU='Next';
                Enabled=NextActionEnabled;
                image=NextRecord;
                InFooterBar=true;
                ApplicationArea=Basic,Suite;

                trigger OnAction();
                begin
                    NextStep(FALSE);
                end;
            }
```

And on "ActionFinish", we need to call the "FinishAction" function:



```
0 references
Action(ActionFinish){
    CaptionML=ENU='Finish';
    Enabled=FinishActionEnabled;
    image=Approve;
    InFooterBar=true;

    trigger OnAction();
    begin
        FinishAction;
    end;
}
```

Task 5: Add image header

High Level Steps

1. Create Groups to hold Media fields
2. Create the local function "LoadTopBanners"
3. Define Header appearance logic

Review the Result

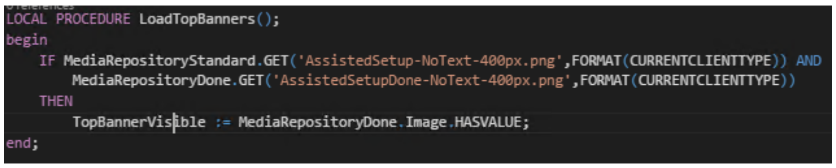
Detailed Steps

1. Create Groups to hold Media fields

- Create global variables:
    - "MediaRepositoryStandard" (Record – "Media Repository")
    - "MediaRespositoryDone" (Record – "Media Repository")
    - "TopBannerVisible" (Boolean)
- On the Wizard page, on top of all existing Groups create two new Groups (same level as the "FirstStep" group)
    - First Goup: Group("MediaStandard")
        - Add field "MediaRepositoryStandard.Image"
    - Second Group: Group("MediaDone")
        - Add field "MediaRepositoryDone.Image"
- Set Group properties for the first group:
    - "Visible": "TopBannerVisible AND NOT SecondStepVisible"
    - "Editable": "FALSE"
- Set Group properties for the second group:
    - "Visible": "TopBannerVisible AND SecondStepVisible"
    - "Editable": "FALSE"

2. Create the local function "LoadTopBanners"

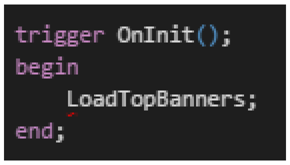
We need this code to get the right pictures out of the media repository. This code can be found on any Wizard page.



```
LOCAL PROCEDURE LoadTopBanners();
begin
    IF MediaRepositoryStandard.GET('AssistedSetup-NoText-400px.png',FORMAT(CURRENTCLIENTTYPE)) AND
        MediaRepositoryDone.GET('AssistedSetupDone-NoText-400px.png',FORMAT(CURRENTCLIENTTYPE))
    THEN
        TopBannerVisible := MediaRepositoryDone.Image.HASVALUE;
end;
```

3. Define Header appearance logic

And to initialize the banners, we need to call the function "OnInit" of the page:



```
trigger OnInit();
begin
    LoadTopBanners;
end;
```

Review the Result

```
page 70050002 "Item Classification Wizard"
{
    PageType = NavigatePage;
    SourceTable = "Item Classification";
    layout
    {
        area(content)
        {
            group("<MediaStandard>")
            {
                Editable=false;
                visible=TopBannerVisible AND NOT SecondStepVisible;
                field("MediaRepositoryStandard.Image";MediaRepositoryStandard.Image)
                {
                    Editable=false;
                    ShowCaption=false;
                }
            }
             group("<MediaDone>"){
                Editable=false;
                visible=TopBannerVisible AND SecondStepVisible;
                field("MediaRepositoryDone.Image";MediaRepositoryDone.Image)
                {
                    Editable=false;
                    ShowCaption=false;
                }
            }
            group(FirstStep){
                Visible=FirstStepVisible;
                group(Welcome){
                    InstructionalTextML=ENU='Welcome';
                }
                group(DataExists){
                    Visible=TablehasData;
                    InstructionalTextML=ENU='there is already data in the Item
Classification Table';
                }
                group(DataEmpty){
                    Visible=NOT TablehasData;
                    InstructionalTextML=ENU='There is no data in the Item
Classification table, so we have provided some default values. Click Next and review
the data.';
                }
            }
            group(SecondStep){
                Visible=SecondStepVisible;
                repeater(ItemClassificationData){
                    Visible=true;
                    field("Code";"Code"){}
                    field(Description;Description){}
                    field("Minimum Sales Count";"Minimum Sales Count"){}
                    field("Warning";"Warning"){}
                }
            }
        }
    }
    actions
    {
            area(processing){
                Action(ActionBack){
                    CaptionML=ENU='Back';
                    Enabled=BackActionEnabled;
                    Image=PreviousRecord;
                    InFooterBar=true;
                    trigger OnAction();
                    begin
                        NextStep(TRUE);
                    end;
                }
                Action(ActionNext){
                    CaptionML=ENU='Next';
                    Enabled=NextActionEnabled;
                    image=NextRecord;
                    InFooterBar=true;
                    trigger OnAction();
                    begin
                        NextStep(FALSE);
                    end;
                }
                Action(ActionFinish){
                    CaptionML=ENU='Finish';
                    Enabled=FinishActionEnabled;
                    image=Approve;
                    InFooterBar=true;
                    trigger OnAction();
                    begin
                        FinishAction;
                    end;
                }
            }
    }
    trigger OnInit();
    begin
        LoadTopBanners;
    end;
    trigger OnOpenPage();
    begin
        Step := Step::First;
        EnableControls;
    end;
    var
        [InDataSet] FirstStepVisible : Boolean;
        [InDataSet] SecondStepVisible : Boolean;
        [InDataSet] TablehasData : boolean;
        Step : Option First,Second;
        [InDataSet] BackActionEnabled : Boolean;
        [InDataSet] NextActionEnabled : Boolean;
        [InDataSet] FinishActionEnabled : Boolean;
        MediaRepositoryStandard : Record "Media Repository";
        MediaRepositoryDone : record "Media Repository";
        [InDataSet] TopBannerVisible : Boolean;
    local procedure ResetControls();
    begin
        FinishActionEnabled := TRUE;
        BackActionEnabled := TRUE;
        NextActionEnabled := TRUE;
        FirstStepVisible := FALSE;
        SecondStepVisible := FALSE;
    end;
    local procedure EnableControls();
    begin
        ResetControls;
        CASE Step of
            Step::First:
                ShowFirstStep;
            Step::Second:
                ShowSecondStep;
        END;
    end;
    local procedure ShowFirstStep();
    var
        ItemClassification: record  "Item Classification";
    begin
        FirstStepVisible := TRUE;
        IF ItemClassification.ISEMPTY THEN BEGIN
            TableHasData := FALSE;
            FinishActionEnabled := FALSE;
            BackActionEnabled := FALSE;
            ItemClassification.InsertDefaultValues;
        END ELSE BEGIN
            TableHasData := TRUE;
            NextActionEnabled := FALSE;
            BackActionEnabled := FALSE;
        END;
    end;
    local procedure ShowSecondStep();
    begin
        SecondStepVisible := TRUE;
        NextActionEnabled := FALSE;
        BackActionEnabled := FALSE;
    end;
    local procedure NextStep (Backwards: Boolean);
    begin
        if Backwards then
            Step := Step - 1
        ELSE
            Step := Step + 1;
        EnableControls;
    end;
    local procedure FinishAction();
    begin
        CurrPage.CLOSE;
    end;
    LOCAL PROCEDURE LoadTopBanners();
    begin
        IF MediaRepositoryStandard.GET('AssistedSetup-NoText-
400px.png',FORMAT(CURRENTCLIENTTYPE)) AND
            MediaRepositoryDone.GET('AssistedSetupDone-NoText-
400px.png',FORMAT(CURRENTCLIENTTYPE))
        THEN
            TopBannerVisible := MediaRepositoryDone.Image.HASVALUE;
    end;
}
```

Exercise 6: Assisted Setup

Exercise Scenario

Task 1: Use the events on the “Aggregated Assisted Setup” to add

your own set to the functionality.

High Level Steps

Create the Event Subscriber “AddItemClassificationWizard_OnRegisterAssistedSetup”

Create the Event Subscriber “UpdateItemClassificationSetupStatus_OnUpdateAssistedSetupStatus”

Create the local function “GetItemClassificationSetupStatus”

Detailed Steps

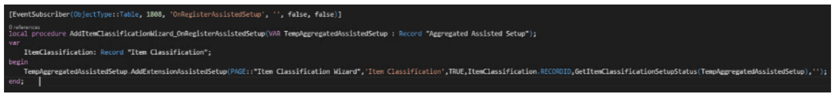
Create the Event Subscriber “AddItemClassificationWizard_OnRegisterAssistedSetup”

- Create a new file “ItemClassificationSubsCodeunit.al”
- In this file, we’ll create a new codeunit:

  Id: 70050003

  "Item Classification Subs"
- Create the Event Subscriber “AddItemClassificationWizard_OnRegisterAssistedSetup”

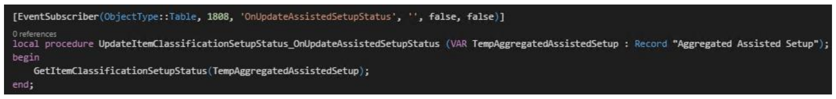
  The business logic should be something like this:



```
[EventSubscriber(ObjectType::Table, 1808, 'OnRegisterAssistedSetup', '', false, false)]
0 references
local procedure AddItemClassificationWizard_OnRegisterAssistedSetup(VAR TempAggregatedAssistedSetup : Record "Aggregated Assisted Setup");
var
    ItemClassification: Record "Item Classification";
begin
    TempAggregatedAssistedSetup.AddExtensionAssistedSetup(PAGE::"Item Classification Wizard",'Item Classification',TRUE,ItemClassification.RECORDID,GetItemClassificationSetupStatus(TempAggregatedAssistedSetup),'');
end;
```

Create the Event Subscriber “UpdateItemClassificationSetupStatus_OnUpdateAssistedSetupStatus”

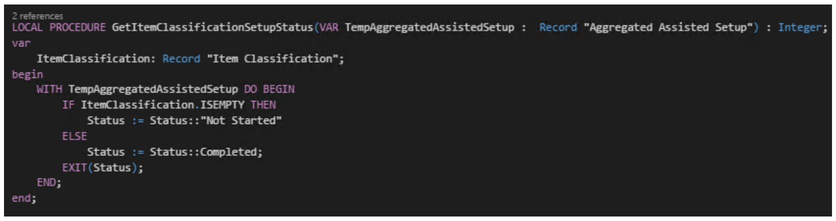
This event will be called to check the status of the particular setup.



```
[EventSubscriber(ObjectType::Table, 1808, 'OnUpdateAssistedSetupStatus', '', false, false)]
0 references
local procedure UpdateItemClassificationSetupStatus_OnUpdateAssistedSetupStatus (VAR TempAggregatedAssistedSetup : Record "Aggregated Assisted Setup");
begin
    GetItemClassificationSetupStatus(TempAggregatedAssistedSetup);
end;
```

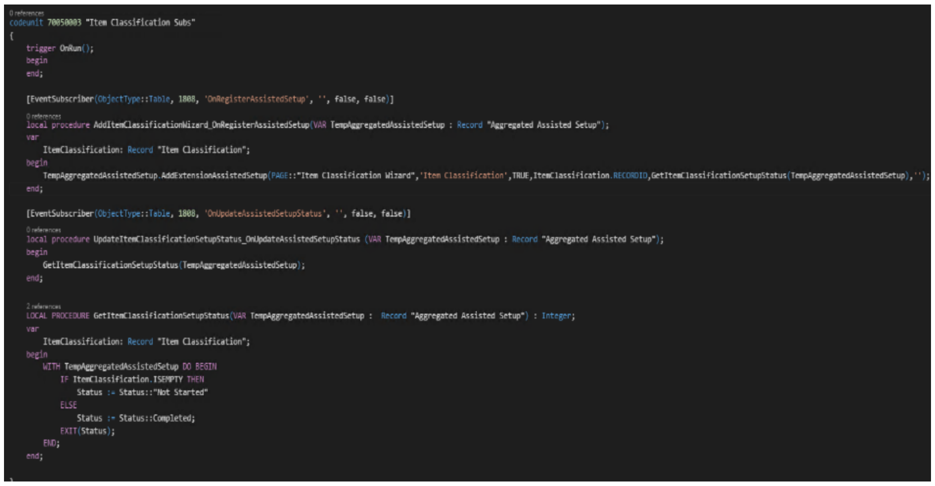
Create the local function “GetItemClassificationSetupStatus”

To get the status of our setup, let’s implement that in a local function like this:



```
2 references
LOCAL PROCEDURE GetItemClassificationSetupStatus(VAR TempAggregatedAssistedSetup :  Record "Aggregated Assisted Setup") : Integer;
var
    ItemClassification: Record "Item Classification";
begin
    WITH TempAggregatedAssistedSetup DO BEGIN
        IF ItemClassification.ISEMPTY THEN
            Status := Status::"Not Started"
        ELSE
            Status := Status::Completed;
        EXIT(Status);
    END;
end;
```

All-in-all, the resulting codeunit looks like this:



```
0 references
codeunit 70050003 "Item Classification Subs"
{
    trigger OnRun();
    begin
    end;

    [EventSubscriber(ObjectType::Table, 1808, 'OnRegisterAssistedSetup', '', false, false)]
    0 references
    local procedure AddItemClassificationWizard_OnRegisterAssistedSetup(VAR TempAggregatedAssistedSetup : Record "Aggregated Assisted Setup");
    var
        ItemClassification: Record "Item Classification";
    begin
        TempAggregatedAssistedSetup.AddExtensionAssistedSetup(PAGE::"Item Classification Wizard",'Item Classification',TRUE,ItemClassification.RECORDID,GetItemClassificationSetupStatus(TempAggregatedAssistedSetup),'');
    end;

    [EventSubscriber(ObjectType::Table, 1808, 'OnUpdateAssistedSetupStatus', '', false, false)]
    0 references
    local procedure UpdateItemClassificationSetupStatus_OnUpdateAssistedSetupStatus (VAR TempAggregatedAssistedSetup : Record "Aggregated Assisted Setup");
    begin
        GetItemClassificationSetupStatus(TempAggregatedAssistedSetup);
    end;


    2 references
    LOCAL PROCEDURE GetItemClassificationSetupStatus(VAR TempAggregatedAssistedSetup :  Record "Aggregated Assisted Setup") : Integer;
    var
        ItemClassification: Record "Item Classification";
    begin
        WITH TempAggregatedAssistedSetup DO BEGIN
            IF ItemClassification.ISEMPTY THEN
                Status := Status::"Not Started"
            ELSE
                Status := Status::Completed;
            EXIT(Status);
        END;
    end;
}
```

Task 2: Test the functionality

High Level Steps

Build the app and test the functionality

Detailed Steps

Build the app and test the functionality

- Change the “launch.json”: remove the “StartupObjectId” setting
- Start the client (use the shortcut on the desktop “Navision_main Web Client”)
- Navigate to the Assisted Setup
- Start the wizard
- Step through all the steps and look at the buttons (enabled/disabled)

Verify the data by calculating a few classifications in the Item list.

Exercise 7: Implement Application Areas

Exercise Scenario

To be able to use the Extension in Dynamics 365, we need to implement Application Areas on all actions and fields of our pages or page extensions.

Task 1: Set up Application Area on Item List

High Level Steps

Set Application Area property for the "Item Classification Code"

Detailed Steps

Set Application Area property for the "Item Classification Code"

- Open "ItemListPageExtension.al"
- On our new field "Item Classification Code", add the property "ApplicationArea"
- Set it to "Basic,Suite"

Task 2: Set up Application Area on Item Classification Wizard

High Level Steps

Set "Application Area" property for all the fields on the "Item Classification Wizard" page

Detailed Steps

Set "Application Area" property for all the fields on the "Item Classification Wizard" page

Apply the "ApplicationArea" property in the Wizard page with values "Basic,Suite" on:

- The Media-fields
- All fields on the repeater
- All the Actions (Back, Next, Finish)

Task 3: Set up Application Area on Item Classification

High Level Steps

Set "Application Area" property for all the fields on the "Item Classifications" page

Detailed Steps

Set "Application Area" property for all the fields on the "Item Classifications" page

- Open "ItemClassificationPage.al"
- On all fields, add the property "ApplicationArea"
- Set it to "Basic,Suite"

Exercise 8: Create Notification for unconfigured Extension.

Exercise Scenario

Create Notification for when the Extension is just installed, but the user hasn't configured it through the assisted setup.

The notification should show a warning like: "You haven't set up the "Item Classification" setup yet. You can do that here" (with "here" a link to the wizard). Obviously, it should only show that if the Extension isn't set up yet (no data in the Item Classification table).

The Notification should get be shown at the these events:

- When you installed the Extension from the Extension Manager
- When you open the Item List or Item Card
- When you use any Item on the Sales Line

Try to implement this Exercise yourself without us providing any steps for it.

Task 1: Try to implement this Exercise yourself without us providing any detailed steps for it.

High Level Steps

- Create a Method Codeunit
- Find and subscribe to the right events to call your new method

Detailed Steps

- Create a Method Codeunit
- Find and subscribe to the right events to call your new method