# EXCEL APPLICATIONS

10 Steps for VBA Developers

Sergey Vaselenko

## EXCELAPPLICATIONS 10 Steps for VBA Developers

Written by Sergey Vaselenko

This e-book shows how to create Excel applications with the SaveToDB add-in and VBA. As a VBA developer, you may get great benefits and create larger applications in less time.

#### Introduction

The SaveToDB add-in allows creating database client applications using Microsoft Excel.

Moreover, application features are being configured in a database and extended using SQL.

So, using the add-in is the best choice to create client applications for database developers.

As a VBA developer, you may create applications using the SaveToDB add-in features or just use it as a library.

You get the fast and reliable platform that solves typical deployment and data layer tasks from-the-box.

Here are the basic steps to create a complete client application using Microsoft Excel:

- 1. Connect to tables, views, and stored procedures
- 2. Configure validation lists
- 3. Configure ribbon parameters
- 4. Translate field, parameter, and object names
- 5. Configure formats, formulas, and table views
- 6. Configure saving changes
- 7. Add cursors and form fields
- 8. Create master-detail forms
- 9. Configure detail windows and task panes
- 10. Configure context and action menus

This book shows how to make these steps using the add-in wizards, SQL codes, and VBA macros.

We start with a new Excel workbook and finish with a ready-to-use application.

You have to download and install the SaveToDB add-in, version 7.2 or higher, at www.savetodb.com.

All features described in this book are available in the free SaveToDB Express edition.

You may download workbooks, SQL and VBA codes, used in this book, at

https://www.savetodb.com/download.php?file=10-steps-for-vba-developers.zip

This book contains an example database for Microsoft SQL Server.

You may also use Oracle Database, IBM DB2, MySQL, PostgreSQL, and others. The steps remain the same.

Best regards,

Sergey Vaselenko

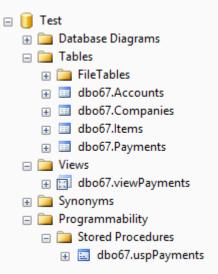
March 20, 2017

#### Table of Contents

Introduction		1
Table of Contents		
Chapter 1.	Example Application	3
Chapter 2.	Excel as Table Editor	5
Chapter 3.	SaveToDB Framework Installer	13
Chapter 4.	Configuration Workbook	16
Chapter 5.	Tables with Foreign Keys	18
Chapter 6.	Query Parameters	22
Chapter 7.	Column Name Translation	25
Chapter 8.	Object Name Translation	28
Chapter 9.	Table Views	29
Chapter 10.	Table Format Wizard	34
Chapter 11.	Framework Query List	36
Chapter 12.	Configuring Views	40
Chapter 13.	Configuring Stored Procedures	43
Chapter 14.	Configuring Saving Changes	45
Chapter 15.	Cursors	50
Chapter 16.	Form Fields	54
Chapter 17.	Master-Details	55
Chapter 18.	Detail Windows and Task Panes	60
Chapter 19.	Context Menus	64
Chapter 20.	Actions Menus	66
Chapter 21.	Creating and Removing Applications	67
Conclusion		68
About the Author		
Appendix 1.	Database Source Code	70

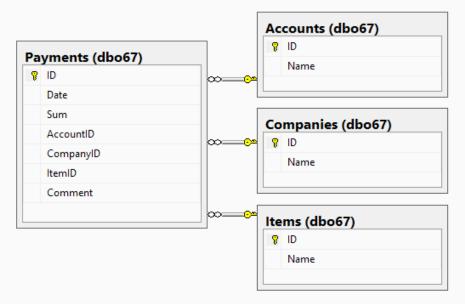
#### Chapter 1. Example Application

We will create an Excel application for a simple database that contains several tables, a view, and a procedure:



The database objects belong to the **dbo67** schema.

The tables have the following relations:



We have three master tables and the dbo67.Payments table with foreign keys.

The dbo67.viewPayments view and dbo67.uspPayments stored procedure select data from the Payments table.

You may find a complete source code with comments and download links in Appendix 1.

Every chapter shows standard SaveToDB features and configuration steps.

Also, it shows VBA codes that can use the described features.

In the downloaded package, you may find a final application that contains complete source codes.

The payments.xlsm workbook contains a single worksheet and creates other sheets using VBA.

Here is the main page:

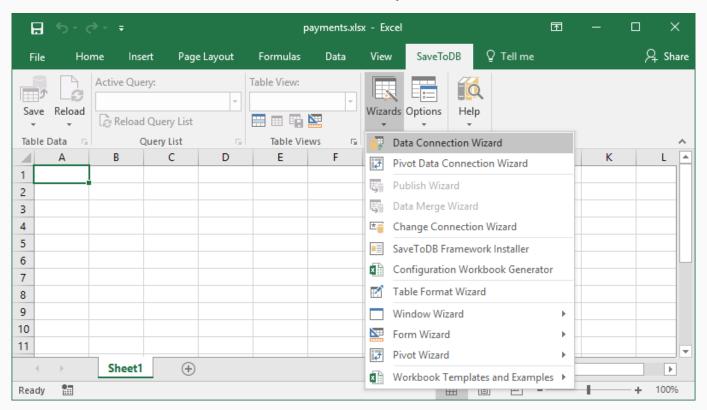
### Evcal Applications

Excel Applications			
10 Steps for VBA Developers			
Chapter 02. Create master table editors	Chapter 13. Change a query using Query List		
Chapter 05. Create the Payments worksheet	Chapter 14. Save changes		
Chapter 06. Configure WHERE parameters	Chapter 15. Add cursors		
Chapter 07. Translate columns	Chapter 15. Add cursor buttons		
Chapter 09. Create table views	Chapter 16. Add form fields		
Chapter 11. Create the Reports worksheet	Chapter 17. Create Company Payments		
_			
Chapter 21. Create the application	Chapter 21. Remove the application		
Copyright © 2017, Sergey Vaselenko	www.savetodb.com/10-steps-for-vba-developers		

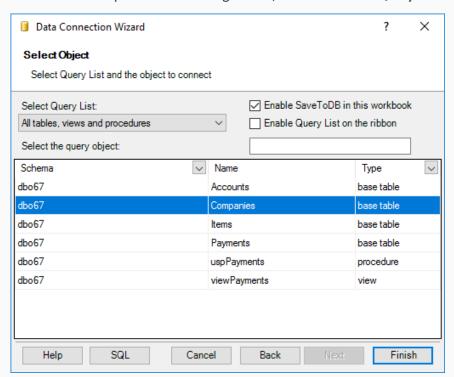
#### Chapter 2. Excel as Table Editor

Let's create a new workbook and save it as payments.xlsx.

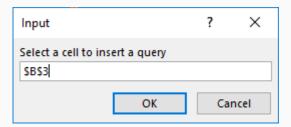
Then let's run the **Data Connect Wizard** and connect to the **Companies** table.



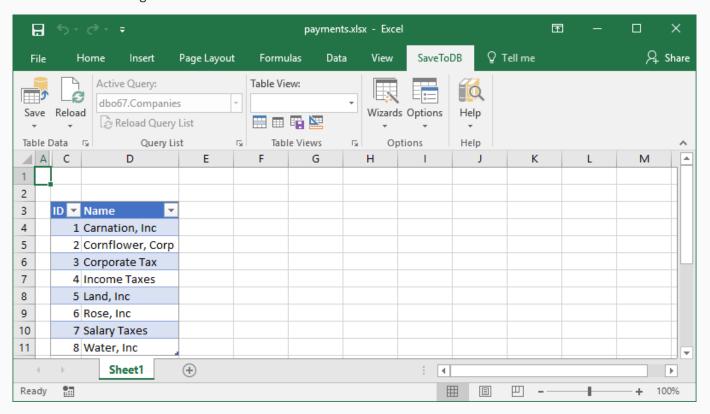
Follow wizard steps. At the following screen, uncheck Enable Query List on the ribbon.



Insert a table at cell B3.



We have the following result:



We may edit data, add and delete rows. Then just click the Save button to save changes.

#### **Creating tables using VBA**

Here is a typical code to insert a connected table:

```
' Inserts the dbo67.Companies table to range B3
' The code demonstrates the simplest way to insert a connected table
' The dummy parameter is used to prevent showing the procedure in a macro list

Sub Chapter02_1_InsertConnectedListObject(ByVal dummy As Boolean)

Dim connString As String

connString = "Provider=SQLOLEDB;Data Source=.\SQLEXPRESS;Initial Catalog=Test"

InsertConnectedListObject Nothing, Range("B3"), connString, "dbo67.Companies", "TABLE"

End Sub
```

Here is a function the actually inserts a ListObject table using the SaveToDB InsertConnectedListObject method:

```
' Inserts a connected ListObject object at the specified range
Function InsertConnectedListObject(ByVal addIn As Object, ByVal r As Range,
   ByVal connString As String, ByVal commandText As String, ByVal objectType As String,
   Optional queryListLocked As Boolean = False) As Boolean
   If addIn Is Nothing Then Set addIn = GetAddIn()
   If addIn Is Nothing Then Exit Function
   InsertConnectedListObject = addIn.InsertConnectedListObject(r, _
           connString, commandText, objectType)
   If Not InsertConnectedListObject Then
       Debug.Print addIn.LastResultMessage
       Exit Function
   End If
   If Not queryListLocked Then Exit Function
   Dim lo As ListObject
   Set lo = r.ListObject
   addIn.QueryLocked(lo) = queryListLocked
End Function
```

In the beginning, we get a SaveToDB add-in variable and then call its methods.

In the end, we get a variable for the created table and change the QueryLocked property.

The following code is used to get the SaveToDB add-in variable:

```
' Returns the SaveToDB addin object
Private AddInObject As Object

Function GetAddIn() As Object

On Error Resume Next

If AddInObject Is Nothing Then

Dim addIn As COMAddIn
Set addIn = Application.COMAddIns("SaveToDB")

If addIn Is Nothing Then Exit Function
Set AddInObject = addIn.Object

End If

Set GetAddIn = AddInObject

End Function
```

You may use it like Call GetAddIn().MethodName(parameters) or like GetAddIn().PropertyName = value.

#### Save and Load using VBA

You may use the following code to reload ListObject table data. This is one add-in method only, Load:

Moreover, you may save data changes using the **Save** method only:

```
' Saves the changes of the ListObject object to a database

Function SaveListObject(ByVal lo As ListObject, Optional ByVal addIn As Object) As Boolean

If addIn Is Nothing Then Set addIn = GetAddIn()

If addIn Is Nothing Then Exit Function

SaveListObject = addIn.Save(lo)

If Not SaveListObject Then

Debug.Print addIn.LastResultMessage
End If

End Function
```

You may use the following macros on the buttons to save and load data of the active tables.

```
' Saves the changes of the active ListObject

Sub Save()

Dim lo As ListObject
Set lo = GetActiveListObject()

If lo Is Nothing Then Exit Sub

Call SaveListObject(lo)

End Sub

' Reloads data of the active ListObject

Sub Load()

Dim lo As ListObject
Set lo = GetActiveListObject()

If lo Is Nothing Then Exit Sub

Call LoadListObject(lo, False)

End Sub
```

This function returns the active ListObject table:

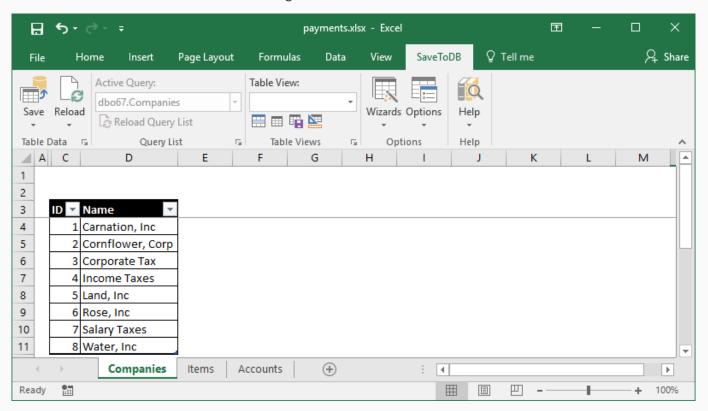
```
' Returns the active ListObject
Function GetActiveListObject() As ListObject
Set GetActiveListObject = ActiveCell.ListObject
If Not GetActiveListObject Is Nothing Then Exit Function
If ActiveSheet.ListObjects.Count <> 1 Then Exit Function
Set GetActiveListObject = ActiveSheet.ListObjects(1)
End Function
```

Let's customize the design:

- 1. Click on a table. Select the **Design** tab. Select the desired design in the **Table Styles** gallery. I prefer White, Table Style Medium 15.
- 2. Right click on the selected design and click **Set as Default**.
- 3. Uncheck **Bunded Rows** in the **Table Style Options** group.
- 4. Select the View tab. Uncheck Gridlines in the Show group.
- 5. Select cell A4 and click Freeze Panes in the Window group.
- 6. Rename the worksheet to Companies.

Repeat the steps for the Items and Accounts tables.

Now we have the workbook that allows editing master tables.



#### Creating master table editors using VBA

The following code allows creating editable tables described in the chapter in a completely new workbook:

```
' Creates master table editors, Chapter 2
Sub Chapter02_2_CreateMasterTableEditors()
   Dim addIn As Object
   Set addIn = GetAddInAndCheck()
   If addIn Is Nothing Then Exit Sub
   Dim wb As Workbook
   Set wb = ActiveWorkbook
    ' Set wb = Workbooks.Add(ActiveWorkbook.FullName)
   Call addIn.InsertAddInSheets(wb)
   Dim connString As String
   connString = GetConnectionString()
   Dim ws As Worksheet
   Set ws = GetOrCreateConnectedWorksheet(wb, _
           "Companies", connString, "dbo67.Companies", "TABLE", True)
   If ws Is Nothing Then Exit Sub
   Call GetOrCreateConnectedWorksheet(wb,
           "Items", connString, "dbo67.Items", "TABLE", True)
   Call GetOrCreateConnectedWorksheet(wb, _
           "Accounts", connString, "dbo67.Accounts", "TABLE", True)
   ws.Select
End Sub
```

The code contains an important call: addIn.InsertAddInSheets(wb)

This method inserts hidden worksheets used by the SaveToDB add-in to make the tables editable.

The GetAddInAndCheck function encapsulates the GetAddIn function and displays the installation error message:

GetOrCreateConnectedWorksheet is a high-level function to create a worksheet with an editable table:

We have discussed the InsertConnectedListObject function above in this chapter.

The following utile function returns an existing worksheet or creates it:

```
' Returns a worksheet, creates if not exists
Function GetOrCreateWorksheet(ByVal wb As Workbook, ByVal sheetName As String) As Worksheet
    Set GetOrCreateWorksheet = GetWorksheet(wb, sheetName)
    If Not GetOrCreateWorksheet Is Nothing Then Exit Function
    Set GetOrCreateWorksheet = AddWorksheet(wb, sheetName)
End Function
```

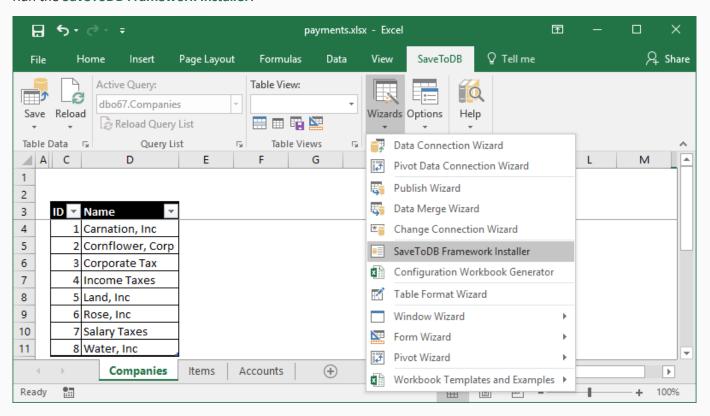
You may find and test a complete code in the payments.xlsm workbook.

#### Chapter 3. SaveToDB Framework Installer

You may use Microsoft Excel as a table editor by default.

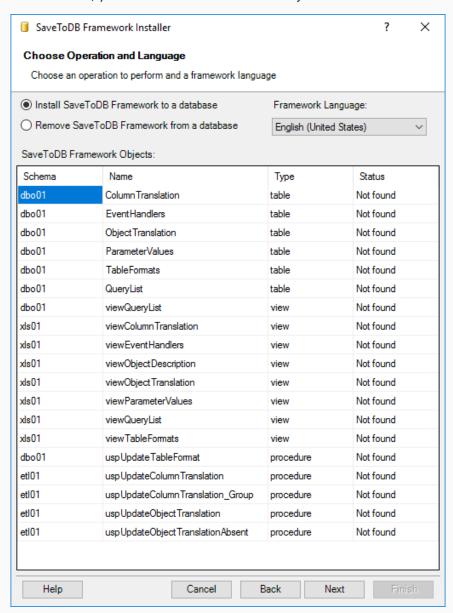
To use advanced features, we have to install the SaveToDB Framework to a database.

Run the SaveToDB Framework Installer:



Follow wizard steps.

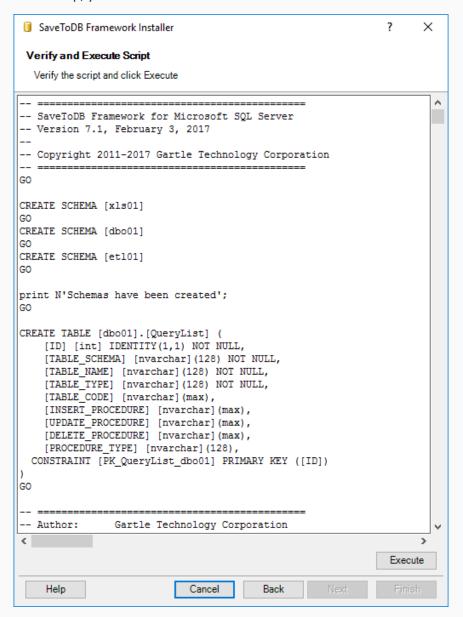
At this screen, you see SaveToDB Framework objects:



These objects allow configuring SaveToDB add-in behavior.

You may remove framework objects using the same wizard. So, this is an entirely safe operation.

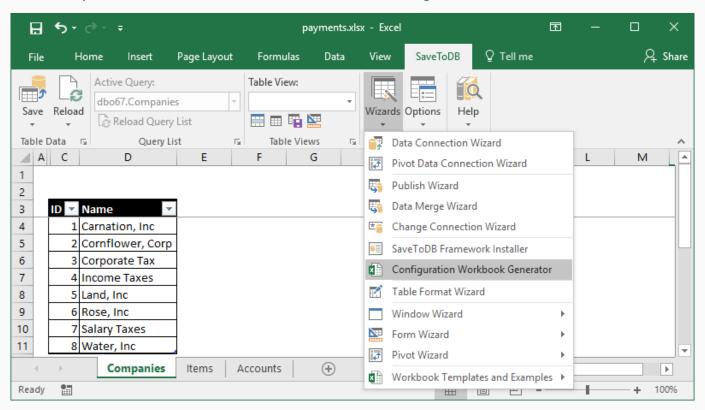
At this step, you see the SaveToDB Framework code. Click **Execute** to install it.



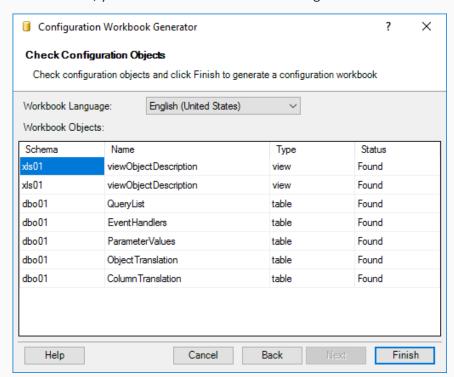
#### Chapter 4. Configuration Workbook

We have added SaveToDB Framework to a database.

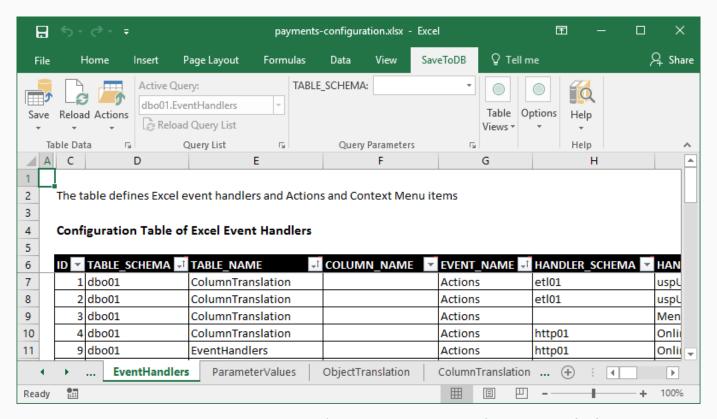
Now we may use Microsoft Excel to edit framework tables. Run Configuration Workbook Generator:



At this screen, you see SaveToDB Framework configuration tables:



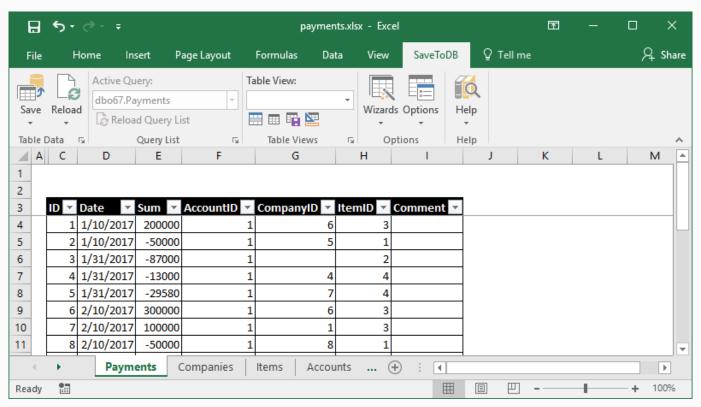
When you click **Finish**, the SaveToDB add-in generates a workbook.



You may generate it again at any time. However, let's save it as payments-configuration.xlsx for further use.

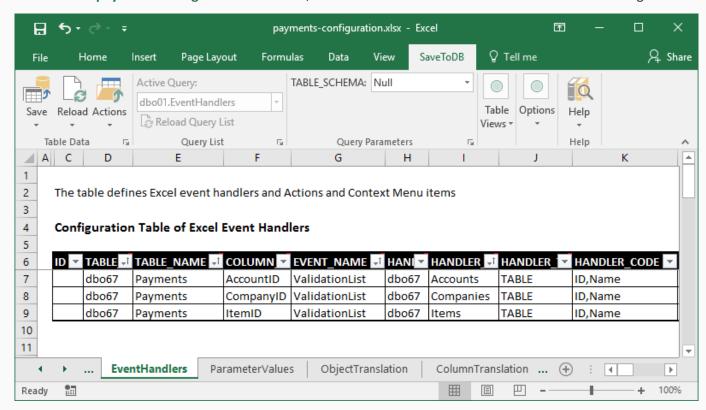
#### Chapter 5. Tables with Foreign Keys

Let's add a worksheet, rename it to Payments, and connect to the Payments table.



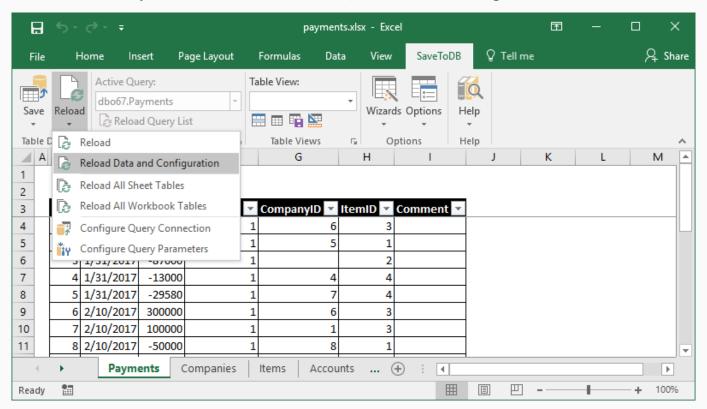
As expected, we see foreign key values. Let's change this.

Switch to the payments-configuration workbook, select the EventHandlers worksheet and add the configuration:

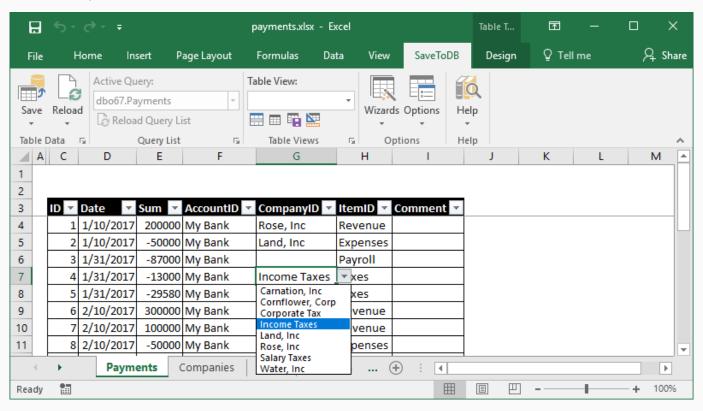


Click the Save button to save the configuration.

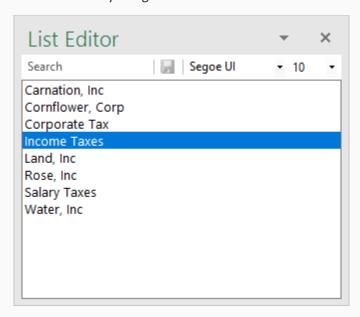
Let's switch to the Payments worksheet and click Reload, Reload Data and Configuration:



The add-in replaces id values with names and adds validation lists:



Moreover, the add-in activates the separate **List Editor** that allows users to select values from large lists in a comfortable way using search.



You may turn on/off the List Editor using the Options, Show List Editor Task Pane option.

Let's change a couple of rows in the table and click the Save, View Save Changes SQL.

In my case, the add-in generates the following SQL commands:

```
Save Change SQL X

File Edit

UPDATE [dbo67].[Payments] SET [Date] = '20170110 00:00:00.000', [Sum] = 200000, [AccountID] = 1, UPDATE [dbo67].[Payments] SET [Date] = '20170110 00:00:00.000', [Sum] = -50000, [AccountID] = 1, GO
```

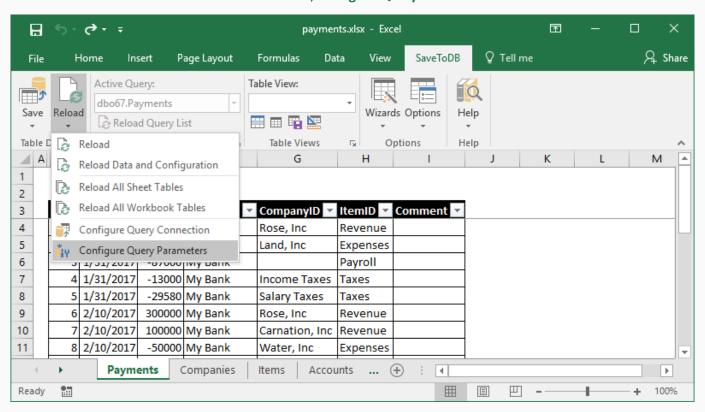
As we may see, the SaveToDB add-in uses id values instead of names as it should be.

#### Creating the Payments table using VBA

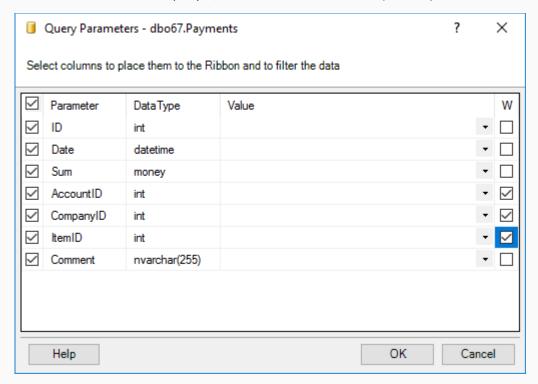
The code, used to create the payments table, is the same as described above.

#### Chapter 6. Query Parameters

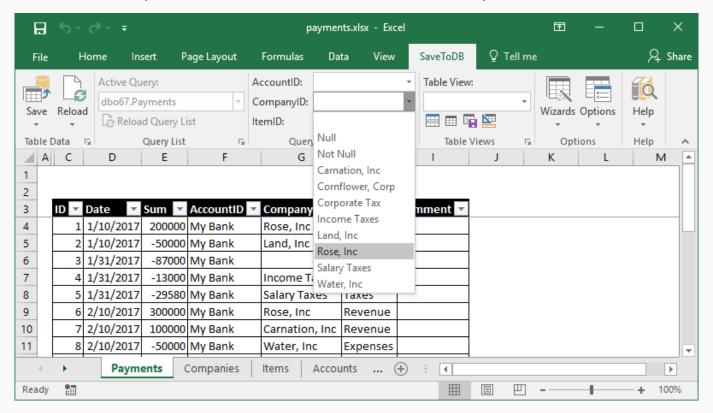
Let's make our table more interactive. Run Reload, Configure Query Parameters:



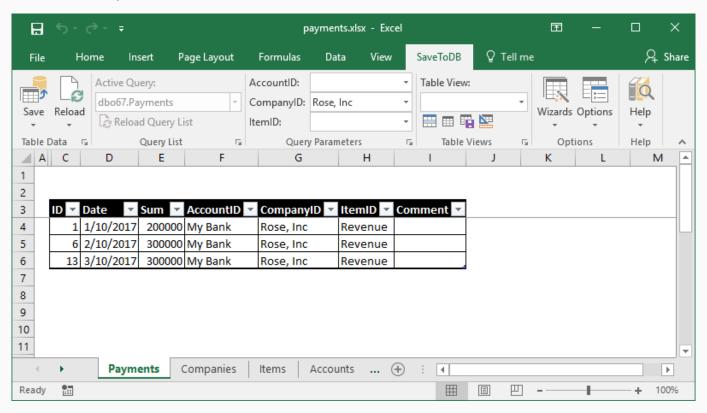
Check the AccountID, CompanyID, and ItemID fields in the **W** (WHERE) column:



The SaveToDB add-in places the selected fields to the ribbon. So, our users may filter data:



Let's choose Rose, Inc.



This feature allows working without auto-filters and loading fewer data.

#### Configuring WHERE fields using VBA

Here is the code used to configure ribbon parameters:

```
' Configures WHERE parameters of the Payments table, Chapter 6
Sub Chapter06_1_SetPaymentsWhereParameters()
    Dim addIn As Object
    Set addIn = GetAddInAndCheck()
    If addIn Is Nothing Then Exit Sub
    Dim ws As Worksheet
    Set ws = GetWorksheet(ActiveWorkbook, "Payments")
    If ws Is Nothing Then
         MsgBox "Worksheet Payments not exists"
         Exit Sub
    End If
    ws.Select
    Dim lo As ListObject
    Set lo = GetFirstListObject(ws)
    If lo Is Nothing Then
         MsgBox "Worksheet Payments does not contain a table"
         Exit Sub
    End If
    addIn.IsRibbonField(lo, "AccountID") = True
addIn.IsRibbonField(lo, "CompanyID") = True
addIn.IsRibbonField(lo, "ItemID") = True
    ' Use ParameterValue to set id values
    ' addIn.ParameterValue(lo, "AccountID") = Nothing
    'addIn.ParameterValue(lo, "CompanyID") = Nothing
     'addIn.ParameterValue(lo, "ItemID") = Nothing
     ' Use ParameterRibbonValue to set name values
    ' addIn.ParameterRibbonValue(lo, "AccountID") = Nothing
' addIn.ParameterRibbonValue(lo, "CompanyID") = Nothing
     'addIn.ParameterRibbonValue(lo, "ItemID") = Nothing
End Sub
```

Use the IsRibbonField property to add or remove ribbon fields of tables and views.

Use the ParameterValue property to set field or stored procedure parameter values using values like ID.

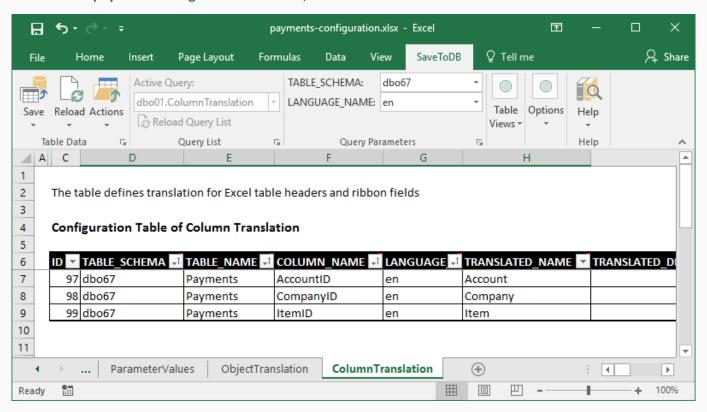
Use the ParameterRibbonValue property to set values translated by ValidationList handlers.

#### Chapter 7. Column Name Translation

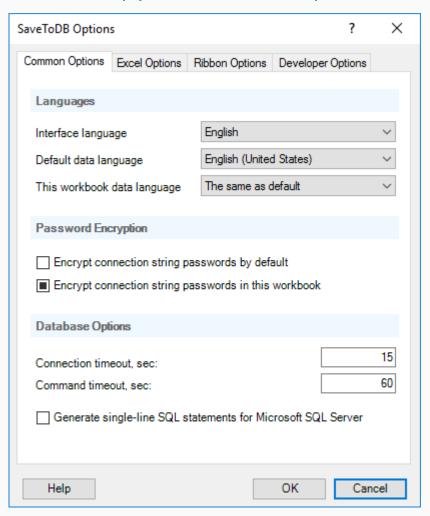
The Payments table shows fields in the table and at the ribbon like AccountID, CompanyID, and ItemID.

Let's change these database field names to business names.

Switch to the payments-configuration workbook, select the ColumnTranslation worksheet and add the data:



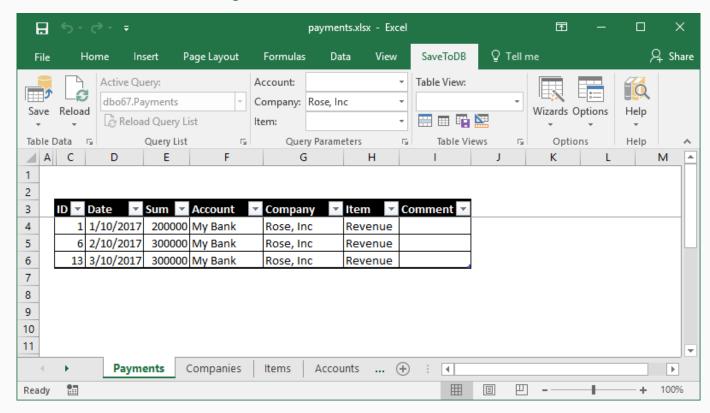
Then switch to the **payments** workbook and click **Options**:



Choose the **Default data language** as English and click **OK**.

Of course, in your applications, you may use any language or even multiple languages.

Click Reload, Reload Data and Configuration.



The add-in shows Account, Company, and Item instead of AccountID, CompanyID, and ItemID.

You may turn off this feature in the **Options** dialog box as described above.

#### **Setting data languages using VBA**

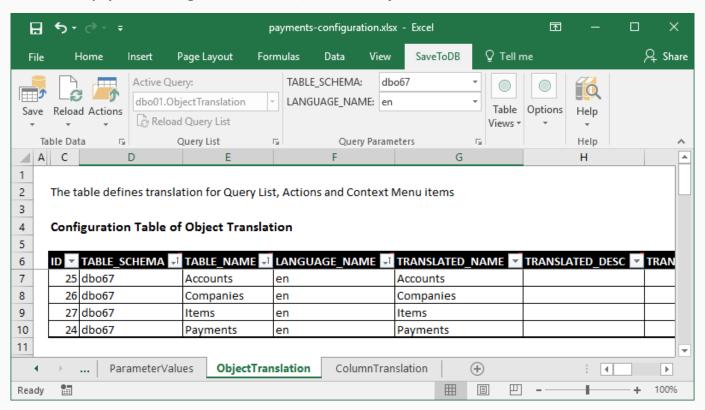
The following code is used to change the data language and reload configurations on all worksheets:

You may use the addIn. **Options** property to change other SaveToDB add-in options too.

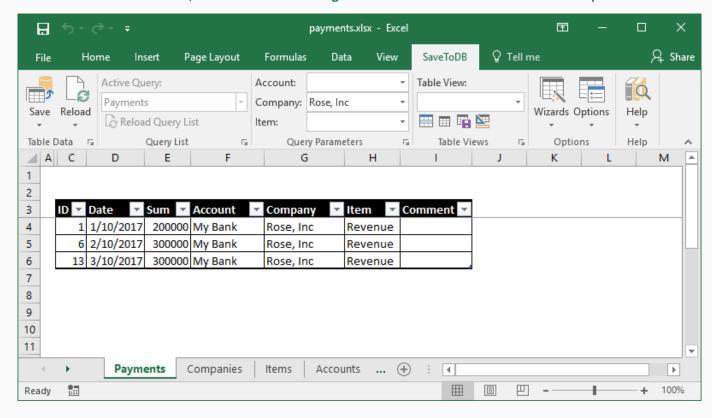
#### Chapter 8. Object Name Translation

We have translated field names. Also, we may translate database object names shown in Excel.

Switch to the payments-configuration workbook, select the ObjectTranslation worksheet and add the data:



Then return and click Reload, Reload Data and Configuration. Now we see business names everywhere.

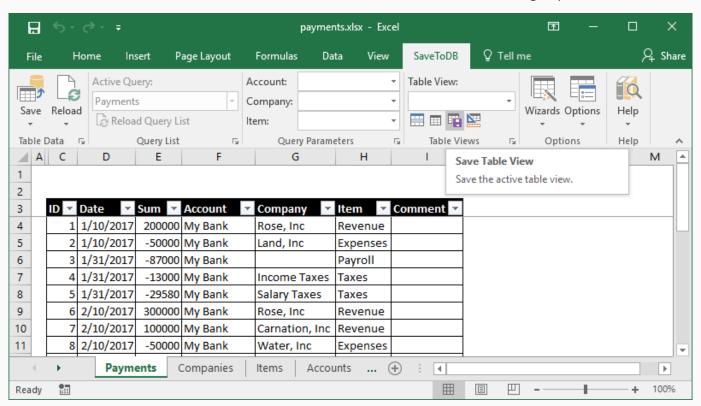


#### Chapter 9. Table Views

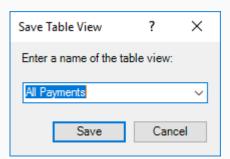
Users often apply different filters to loaded data, hide and unhide columns, sort in various ways, etc.

The SaveToDB add-in may help to save such user views and even share them with colleagues.

Let's remove all WHERE filters and click the Save Table View button in the Table Views group:

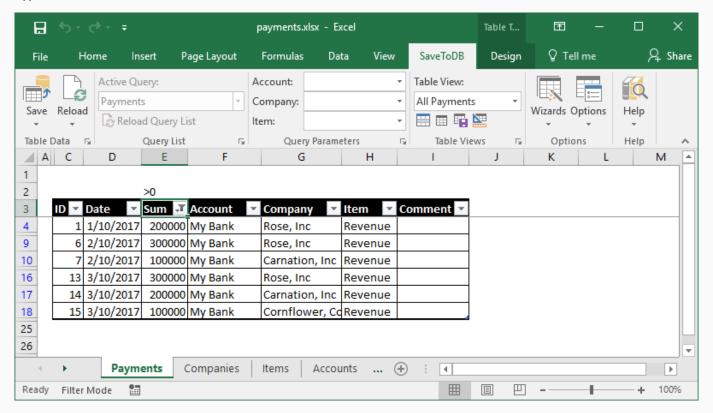


Type All Payments and click Save.



We see the name of the current view, All Payments, in the **Table View** field.

Type >0 in cell E2.

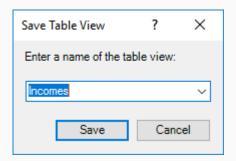


The add-in applies the filter to the Sum column.

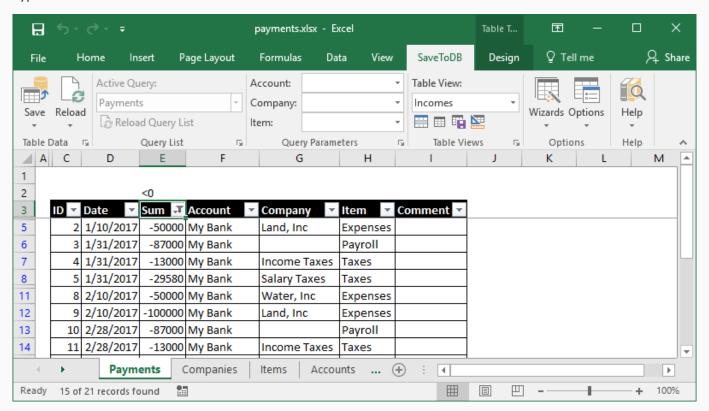
This is a reason why it is better to insert tables at cell B3.

Users may use row 2 (as a row over the table) as auto-filters. Also, they may place formulas in row 1.

Let's continue and save the view as Incomes (click the Save Table View button again):

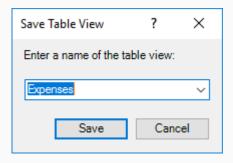


#### Type < 0 in cell E2.

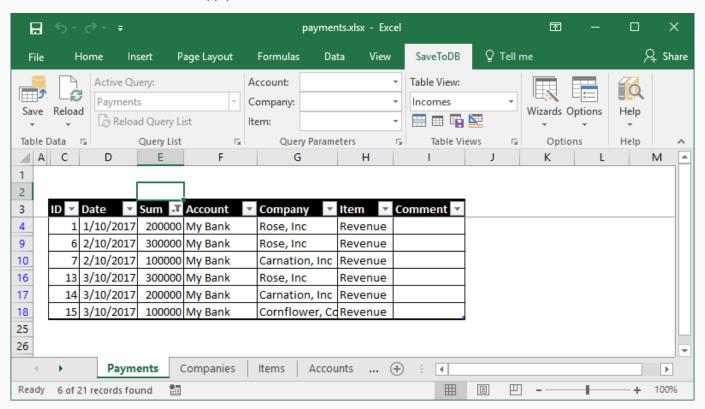


The add-in applies the new filter to the Sum column.

Save the view as **Expenses**.



Remove the filter in cell E2 and apply the Incomes view:



As we may expect, the add-in applies the saved filter to the Sum column.

I am sure, your users will be happy, and you will have fewer requests for new small database views.

#### Creating table views using VBA

The following code creates table views described in the chapter:

```
' Creating table views, Chapter 9
Sub Chapter09_1_CreateTableViews()
    Dim addIn As Object
    Set addIn = GetAddInAndCheck()
   If addIn Is Nothing Then Exit Sub
    Dim ws As Worksheet
    Set ws = GetWorksheet(ActiveWorkbook, "Payments")
    If ws Is Nothing Then
        MsgBox "Worksheet Payments not exists"
        Exit Sub
    End If
    ws.Select
    Dim lo As ListObject
    Set lo = GetFirstListObject(ws)
    If lo Is Nothing Then
        MsgBox "Worksheet Payments does not contain a table"
        Exit Sub
    End If
    Dim cell As Range
    Set cell = ws.Range("E2")
    Call addIn.SaveTableView(lo, "All Payments")
    cell.Value = ">0"
    Call addIn.SaveTableView(lo, "Incomes")
    cell.Value = "<0"
    Call addIn.SaveTableView(lo, "Expenses")
    cell.ClearContents
    Call addIn.ApplyTableView(lo, "Incomes")
End Sub
```

Mainly, you will use the **SaveTableView** and **ApplyTableView** methods.

Also, you may use the auto-filter row over the tables to filter data in an easy way.

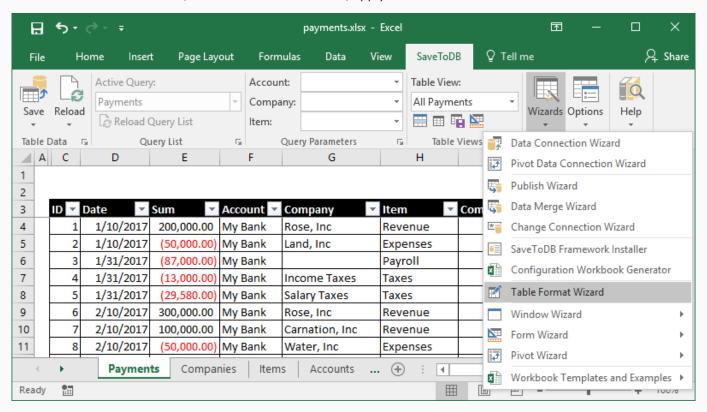
#### Chapter 10. Table Format Wizard

We have formatted tables and added views in the previous steps in the payments.xlsx workbook.

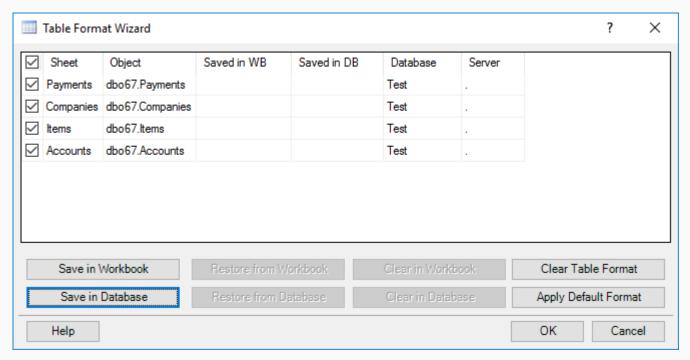
If a user connects to a database from a new workbook, he will have Excel defaults.

We can fix this publishing table formats and views to a database using Table Format Wizard.

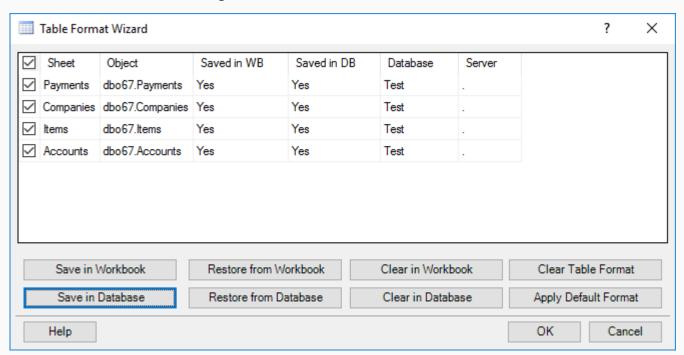
Let's format the Sum column, set default column widths, apply the default table view and run the wizard:



In the wizard, select all tables and click the Save in Database button:



The wizard saves formats and changes its state:



Now, users will get the same formats, views, and formulas of the tables when they connect to a database.

Use the wizard to republish new views later.

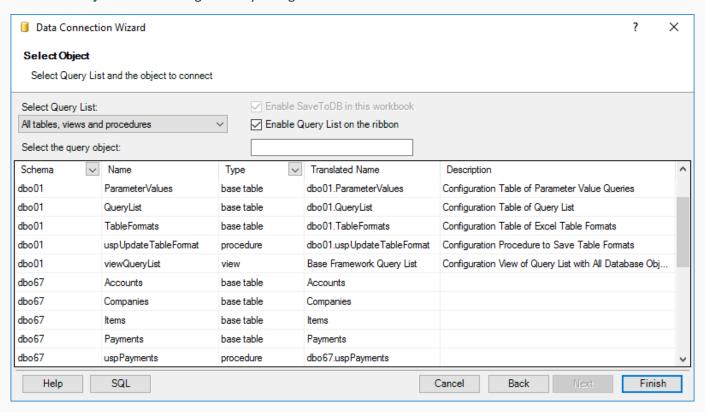
Users may use the **Restore from Database** button to reload the updated views.

SaveToDB 7.2 has no methods to manage formats using VBA. You may do this using the wizard only.

## Chapter 11. Framework Query List

Let's create a worksheet, name it as **Reports** and connect to the **viewPayments** view.

The **Select Object** screen has significantly changed since the last connection:

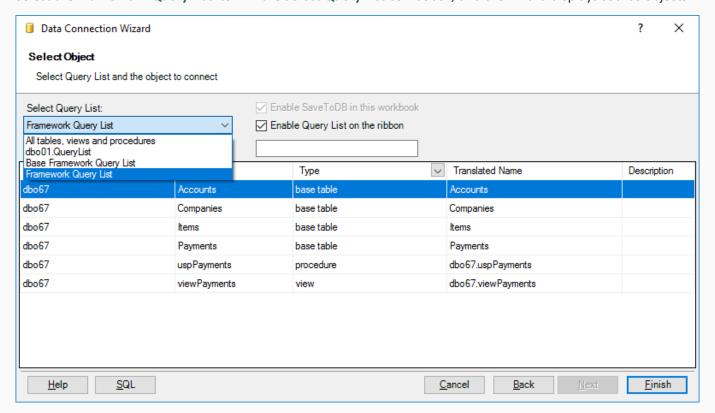


It includes much more objects and additional columns like **Translated Name** and **Description**.

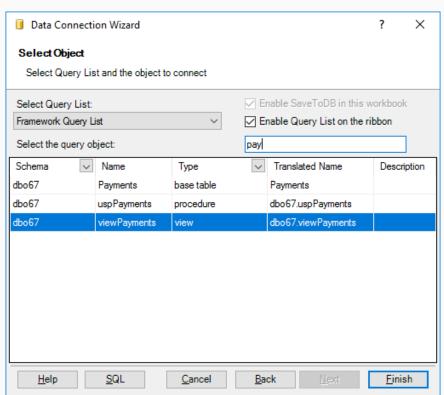
First of all, **Translated Name** and **Description** are shown as we added the SaveToDB Framework, added the object translation, and selected English in the **Options**. This feature helps users to understand database objects better.

The second, the wizard shows installed SaveToDB Framework objects.

Select the Framework Query List item in the Select Query List combobox, and the wizard displays source objects:

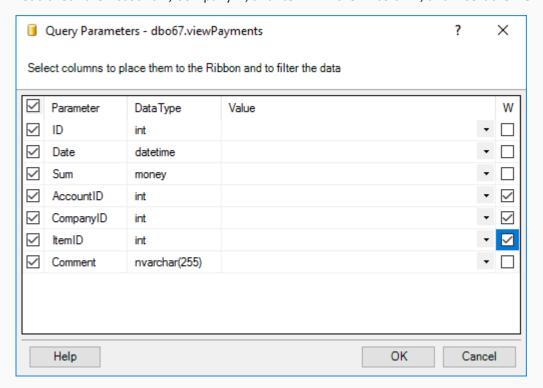


You may filter objects, also. Type pay, for example.

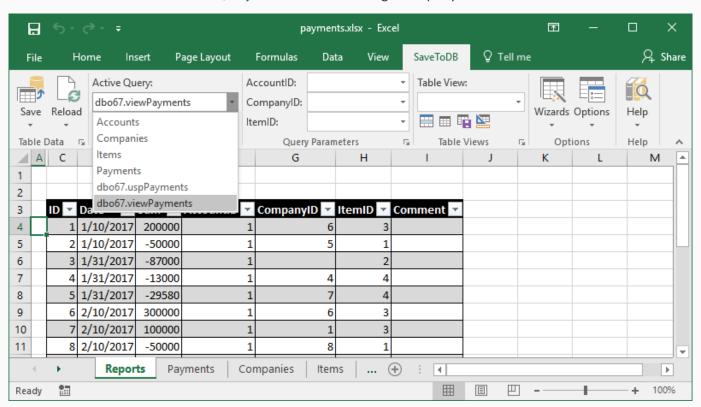


Let's select the dbo67.viewPayments view and leave Enable Query List on the ribbon checked.

Let's check the AccountID, CompanyID, and ItemID in the W column, and insert the view at cell B3.



We see a new feature. The ribbon **Query List** allows us to change the query.



This is a very useful feature. Users may use a few worksheets to work with multiple database objects.

Moreover, when developers add new objects to a database, users just reload the query list and can connect.

#### Creating the Reports worksheet using VBA

The following code is similar to the previous chapter codes:

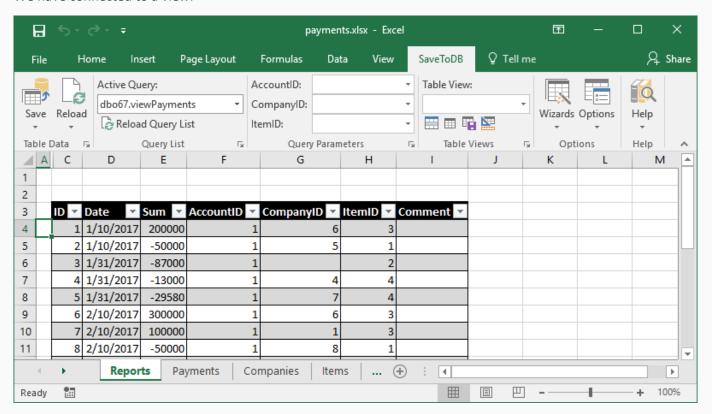
```
' Creates the Reports worksheet, Chapter 11
Sub Chapter11_1_CreateReportsWorksheet()
    Dim addIn As Object
    Set addIn = GetAddInAndCheck()
    If addIn Is Nothing Then Exit Sub
    Dim wb As Workbook
    Set wb = ActiveWorkbook
    Call addIn.InsertAddInSheets(wb)
    Dim connString As String
    connString = GetConnectionString()
    Dim ws As Worksheet
    Set ws = GetOrCreateConnectedWorksheet(wb, _
              "Reports", connString, "dbo67.viewPayments", "VIEW", True)
    If ws Is Nothing Then Exit Sub
    ws.Select
    Dim lo As ListObject
    Set lo = GetFirstListObject(ws)
    If lo Is Nothing Then Exit Sub
    addIn.QueryList(lo) = "xls01.viewQueryList"
    addIn.QueryLocked(lo) = False
    Call addIn.ReloadQueryList(lo)
    addIn.IsRibbonField(lo, "AccountID") = True
addIn.IsRibbonField(lo, "CompanyID") = True
addIn.IsRibbonField(lo, "ItemID") = True
     ' Use ParameterValue to set id values
     'addIn.ParameterValue(lo, "AccountID") = Nothing
    'addIn.ParameterValue(lo, "CompanyID") = Nothing
     'addIn.ParameterValue(lo, "ItemID") = Nothing
     ' Use ParameterRibbonValue to set name values
     ' addIn.ParameterRibbonValue(lo, "AccountID") = Nothing ' addIn.ParameterRibbonValue(lo, "CompanyID") = Nothing ' addIn.ParameterRibbonValue(lo, "ItemID") = Nothing
End Sub
```

It calls two new methods used to set the query list object and reload it:

```
addIn.QueryList(Io) = "xls01.viewQueryList"
Call addIn.ReloadQueryList(Io)
```

# Chapter 12. Configuring Views

#### We have connected to a view:



#### Good news:

- 1. We can save changes (as the view is updateable).
- 2. We can use ribbon parameters to filter data.

#### Bad news:

- 1. We see foreign key values instead of names in columns and ribbon parameters.
- 2. We see AccountID, CompanyID, and ItemID in columns and ribbon parameters.
- 3. We see the dbo67.viewPayments database name in the Query List.
- 4. The table is unformatted.
- 5. There are no predefined table views.

Let's fix the "bad news."

### Replacing foreign keys values with names

Switch to the **payments-configuration** workbook, select the **EventHandlers** worksheet, copy and paste three rows of the **Payment** table, and change Payments to viewPayments in the **TABLE\_NAME** field. Click **Save**.

ID 🔻	TABLE	TABLE_NAME 🗐	COLUMN 🔻	EVENT_NAME 💐	HANI ▼	HANDLER_ ↓↑	HANDLER 🔻	HANDLER_CODE 🔽
26	dbo67	Payments	AccountID	ValidationList	dbo67	Accounts	TABLE	ID,Name
27	dbo67	Payments	CompanyID	ValidationList	dbo67	Companies	TABLE	ID,Name
28	dbo67	Payments	ItemID	ValidationList	dbo67	Items	TABLE	ID,Name
29	dbo67	viewPayments	AccountID	ValidationList	dbo67	Accounts	TABLE	ID,Name
30	dbo67	viewPayments	CompanyID	ValidationList	dbo67	Companies	TABLE	ID,Name
31	dbo67	viewPayments	ItemID	ValidationList	dbo67	Items	TABLE	ID,Name

It is easy. We just copied the existing Payments table configuration.

### **Translating column names**

Select the **ColumnTranslation** worksheet, copy and paste three rows of the **Payment** table, and change Payments to viewPayments in the **TABLE\_NAME** field. Click **Save**.

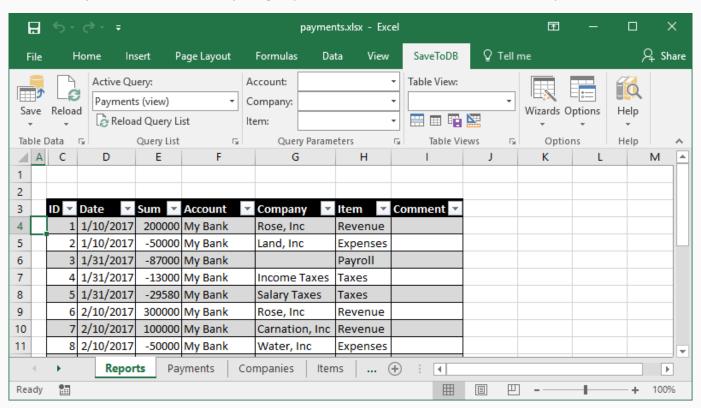
ID 🔻	TABLE_SCHEMA 🚽	TABLE_NAME 🖈	COLUMN_NAME 🖈	LANGUAGE 🚅	TRANSLATED_NAME 🔻	TRANSLATED_DESC 🔻
97	dbo67	Payments	AccountID	en	Account	
98	dbo67	Payments	CompanyID	en	Company	
99	dbo67	Payments	ItemID	en	Item	
100	dbo67	viewPayments	AccountID	en	Account	
101	dbo67	viewPayments	CompanyID	en	Company	
102	dbo67	viewPayments	ItemID	en	Item	

### **Translating object names**

Select the **ObjectTranslation** worksheet and add the translation for **viewPayments** and **uspPayments**:

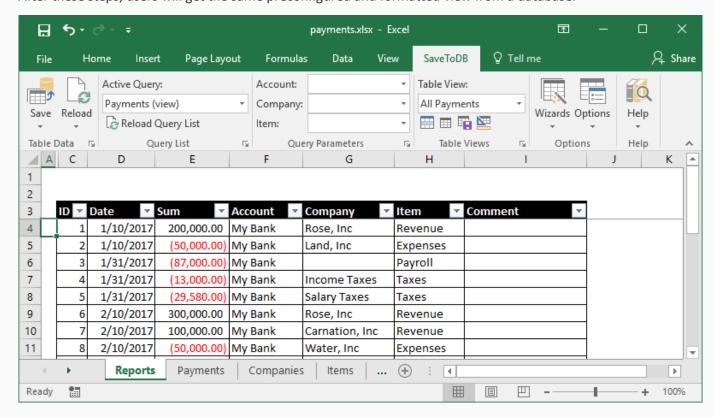
ID 🔻	TABLE_SCHEMA 💐	TABLE_NAME 🖈	<b>↓</b> Î	TRANSLATED_NAME 🔻	TRANSLATED_DESC 🔻	TRANSLATED_COMMENT
25	dbo67	Accounts	en	Accounts		
26	dbo67	Companies	en	Companies		
27	dbo67	Items	en	Items		
24	dbo67	Payments	en	Payments		
29	dbo67	uspPayments	en	Payments (sp)		
28	dbo67	viewPayments	en	Payments (view)		

Now we can switch to the **payments** workbook, click the **Reload**, **Reload Data and Configuration** button, and the **Reload Query List** button in the **Query List** group. We see that we have fixed the first three points.



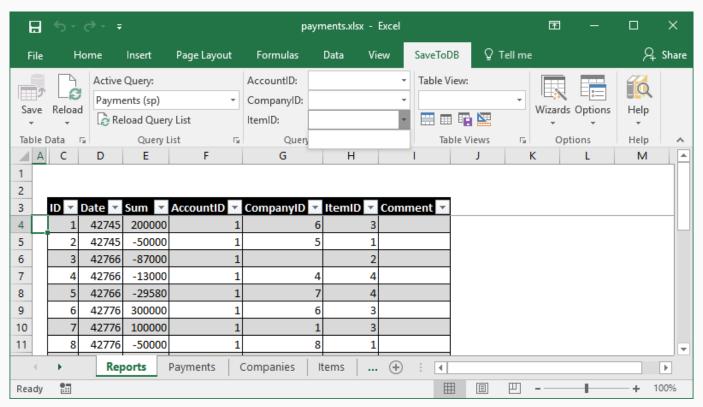
Now we can format the table, add the required table views, and save them to a database using the **Table Format Wizard**, as described in the previous topics.

After these steps, users will get the same preconfigured and formatted view from a database.



# Chapter 13. Configuring Stored Procedures

Let's select the Payments (sp) object in the Query List, a shorter way of Data Connection Wizard.



We see the same points as for the view in the previous chapter:

- Foreign key values instead of names in columns and ribbon parameters;
- AccountID, CompanyID, and ItemID in columns and ribbon parameters;
- Unformatted table;
- No predefined table views.

#### Plus:

- 1. The ribbon parameters have no value lists.
- 2. We can't save the changes.

We already know the way to solve the first group issues.

Here is a configuration for the validation lists to replace id to names:

ID 🔻	TABLE →1	TABLE_NAME 🖈	<b>COLUMN</b>	EVENT_NAME 🗐	HANI 🔻	HANDLER_ ↓1	HANDLER_ ~	HANDLER_CODE 🔻
26	dbo67	Payments	AccountID	ValidationList	dbo67	Accounts	TABLE	ID,Name
27	dbo67	Payments	CompanyID	ValidationList	dbo67	Companies	TABLE	ID,Name
28	dbo67	Payments	ItemID	ValidationList	dbo67	Items	TABLE	ID,Name
32	dbo67	uspPayments	AccountID	ValidationList	dbo67	Accounts	TABLE	ID,Name
33	dbo67	uspPayments	CompanyID	ValidationList	dbo67	Companies	TABLE	ID,Name
34	dbo67	uspPayments	ItemID	ValidationList	dbo67	Items	TABLE	ID,Name
29	dbo67	viewPayments	AccountID	ValidationList	dbo67	Accounts	TABLE	ID,Name
30	dbo67	viewPayments	CompanyID	ValidationList	dbo67	Companies	TABLE	ID,Name
31	dbo67	viewPayments	ItemID	ValidationList	dbo67	Items	TABLE	ID,Name

Here is a configuration to change database column and parameter names to business ones:

ID 🔻	TABLE_SCHEMA 💐	TABLE_NAME 🗐	COLUMN_NAME 🗐	LANGUAGE 💐	TRANSLATED_NAME 🔻	TRANSLATED_DESC 🔻
97	dbo67	Payments	AccountID	en	Account	
98	dbo67	Payments	CompanyID	en	Company	
99	dbo67	Payments	ItemID	en	Item	
103	dbo67	uspPayments	AccountID	en	Account	
104	dbo67	uspPayments	CompanyID	en	Company	
105	dbo67	uspPayments	ItemID	en	Item	
100	dbo67	viewPayments	AccountID	en	Account	
101	dbo67	viewPayments	CompanyID	en	Company	
102	dbo67	viewPayments	ItemID	en	Item	

## **Configuring ribbon parameters**

To configure ribbon parameters, use the **ParameterValues** table of the **payments-configuration** workbook:

ID 🔻	TABLE ~	TABLE_NAME 🔻	PARAMETER_NAME	SELECT_SCHEMA	SELECT_NAME 🔻	SELECT_TYPE 🔻	SELECT_CODE 🔻
1	dbo67	uspPayments	AccountID	dbo67	Accounts	TABLE	ID,Name
2	dbo67	uspPayments	CompanyID	dbo67	Companies	TABLE	ID,Name
3	dbo67	uspPayments	ItemID	dbo67	Items	TABLE	ID,Name

The configuration is like the known **EventHandlers** table.

Thus, we use master tables to select ID and Name pairs for the ribbon parameters of the stored procedure.

### **Configuring saving changes**

Let's discuss this in the next important chapter.

### **Changing query objects using VBA**

We have learned that the ribbon Query List allows changing query objects.

You may do this using the **QueryObject** method:

```
' Changing queries using Query List, Chapter 13
Sub Chapter13_1_ChangeQueryObject()
...
Call addIn.ReloadQueryList(lo)
addIn.QueryObject(lo) = "dbo67.uspPayments"
End Sub
```

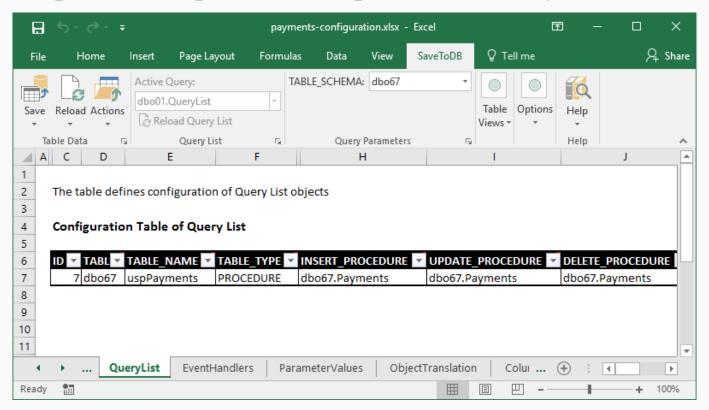
Be sure that the add-in loaded the query list configuration before.

You may use the ReloadQueryList method to reload it.

# Chapter 14. Configuring Saving Changes

### Saving changes to a single table

If we need to save changes to a single underlying table, we may specify the target table in the **INSERT\_PROCEDURE**, update\_procedure, and delete\_procedure fields of the QueryList table:



In this example, this is a right case as the procedure selects data from a single table.

### Saving changes to multiple tables

You may save changes from an Excel table to multiple database tables.

However, this requires using stored procedures or SQL codes.

Let's create such procedures for our example.

Here is the procedure used to insert new rows:

```
CREATE PROCEDURE [dbo67].[uspPayments_insert]
    @Date datetime = NULL
    , @Sum money = NULL
    , @AccountID int = NULL
    , @CompanyID int = NULL
    , @ItemID int = NULL
    , @Comment nvarchar(255) = NULL
AS
BEGIN
SET NOCOUNT ON
INSERT INTO dbo67.Payments
    ( [Date]
    , [Sum]
    , AccountID
    , CompanyID
    , ItemID
    . Comment
VALUES
    ( @Date
    , @Sum
    , @AccountID
    , @CompanyID
    , @ItemID
    , @Comment
END
GO
```

You may see that the procedure has the parameters named as the selected column names.

So, the add-in just calls the procedure passing values from a new row.

You may implement any logic in stored procedures. This procedure just inserts a row into the **Payments** table.

Here is the code of the update procedure:

```
CREATE PROCEDURE [dbo67].[uspPayments update]
      @ID int
    , @Date datetime = NULL
    , @Sum money = NULL
    , @AccountID int = NULL
    , @CompanyID int = NULL
    , @ItemID int = NULL
    , @Comment nvarchar(255) = NULL
AS
BEGIN
SET NOCOUNT ON
UPDATE dbo67.Payments
SET
    [Date] = @Date
    , [Sum] = @Sum
    , AccountID = @AccountID
    , CompanyID = @CompanyID
    , ItemID = @ItemID
    , Comment = @Comment
WHERE
    ID = @ID
END
GO
```

You may see that the logic is the same. The procedure declares and uses parameters with the column names.

The update procedure has the @ID parameter as the updated row exists. The insert procedure has not.

Here is the code of the delete procedure:

```
CREATE PROCEDURE [dbo67].[uspPayments_delete]
    @ID int

AS

BEGIN

SET NOCOUNT ON

DELETE dbo67.Payments
WHERE
    ID = @ID

END
GO
```

In most cases, delete procedures use primary key column values only.

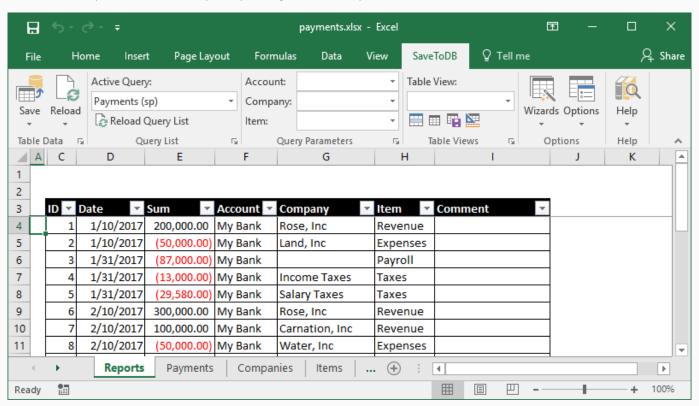
Now, we may replace the configuration created in the previous step

ID TABL	TABLE_NAME 🔻	TABLE_TYPE 🔻	INSERT_PROCEDURE 🔻	UPDATE_PROCEDURE 🔻	DELETE_PROCEDURE 🔻	PROCEDURE_TYPE 🔻
7 dbo67	uspPayments	PROCEDURE	dbo67.Payments	dbo67.Payments	dbo67.Payments	TABLE

to a new configuration with stored procedures:



After these steps, we have a completely configured stored procedure.



Let's change a couple of rows and check the generated SQL code using the Save, View Save Change SQL button.

```
Save Change SQL X

File Edit

EXEC [dbo67].[uspPayments_update] @ID = 1, @Date = '20170110 00:00:00.000', @Sum = 200000, @Accor

EXEC [dbo67].[uspPayments_update] @ID = 2, @Date = '20170110 00:00:00.000', @Sum = -50000, @Accor

GO
```

As we see, the add-in generated EXEC commands.

Using stored procedures to save changes is a common way. In this case, we have four procedures like these:

- 1. dbo67.uspPayments
- dbo67.uspPayments\_insert
- 3. dbo67.uspPayments\_update
- 4. dbo67.uspPayments\_delete

If you use the \_insert, \_update, and \_delete name convention, the add-in links edit procedures automatically, and you may skip adding the configuration to the QueryList table.

### Saving changes using VBA

We have used the **Save** method to save data changes for tables in the previous chapters.

We use it also to save data changes for data loaded from stored procedures in the same manner:

```
' Saving changes, Chapter 14

Sub Chapter14_1_SaveChanges()

Dim addIn As Object
Set addIn = GetAddInAndCheck()

If addIn Is Nothing Then Exit Sub

Dim lo As ListObject
Set lo = GetActiveListObject()

If lo Is Nothing Then Exit Sub

If Not addIn.Save(lo) Then
Debug.Print addIn.LastResultMessage
End If

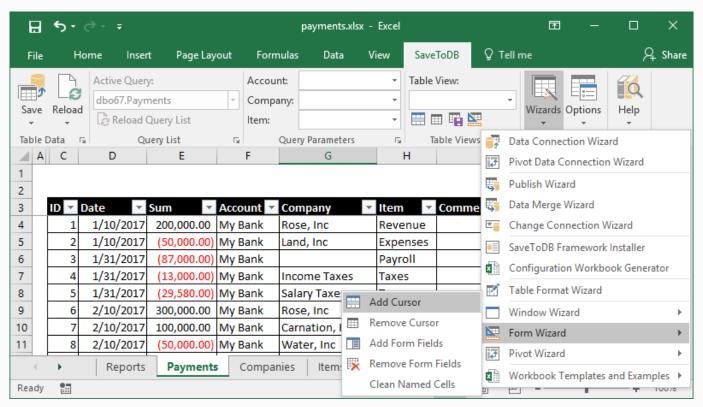
End Sub
```

As a VBA developer, you do not care about the details.

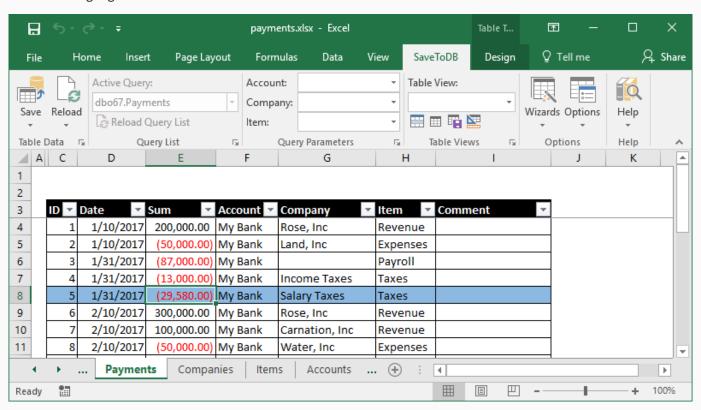
If the saving changes is configured in a database, you just call the **Save** method.

## Chapter 15. Cursors

Let's select the Payments worksheet and click Wizards, Form Wizard, Add Cursor:



The add-in highlights the active table row:



### Adding cursors using VBA

Adding cursors is easy. Just call the **AddTableCursor** method:

```
' Adding cursors, Chapter 15
Sub Chapter15_1_AddCursors()
    Dim addIn As Object
    Set addIn = GetAddInAndCheck()
   If addIn Is Nothing Then Exit Sub
    Dim ws As Worksheet
    Set ws = GetWorksheet(ActiveWorkbook, "Payments")
    If ws Is Nothing Then
        MsgBox "Worksheet Payments not exists"
        Exit Sub
    End If
    ws.Select
    Dim lo As ListObject
    Set lo = GetActiveListObject()
    If lo Is Nothing Then
        MsgBox "Worksheet Payments does not contain a table"
        Exit Sub
    End If
    ' Call addIn.RemoveTableCursor(lo)
    Call addIn.AddTableCursor(lo)
End Sub
```

## Adding cursor buttons using VBA

You may use the following macros to create buttons to load and save data, and to move cursors:

```
' Adds Save and Load buttons

Sub AddSaveAndLoadButtons(ByVal ws As Worksheet)

If ws.Buttons.Count > 0 Then Exit Sub

With ws.Buttons.Add(204, 4, 72, 18)
.Name = "Save"
.Caption = "Save"
.OnAction = "Save"
End With

With ws.Buttons.Add(278, 4, 72, 18)
.Name = "Load"
.Caption = "Load"
.OnAction = "Load"
End With

End Sub
```

```
' Adds cursors buttons
Sub AddCursorButtons(ByVal ws As Worksheet)
    If ws.Buttons.Count > 2 Then Exit Sub
    With ws.Buttons.Add(360, 4, 36, 18)
        .Name = "MoveToFirst"
        .Caption = " <"
        .OnAction = "MoveToFirst"
    End With
    With ws.Buttons.Add(398, 4, 36, 18)
        .Name = "MoveToPrev"
        .Caption = "<"
        .OnAction = "MoveToPrev"
    End With
    With ws.Buttons.Add(436, 4, 36, 18)
        .Name = "MoveToNext"
        .Caption = ">"
        .OnAction = "MoveToNext"
    End With
    With ws.Buttons.Add(474, 4, 36, 18)
        .Name = "MoveToLast"
        .Caption = ">|"
        .OnAction = "MoveToLast"
    End With
End Sub
```

Below are the macros of the buttons.

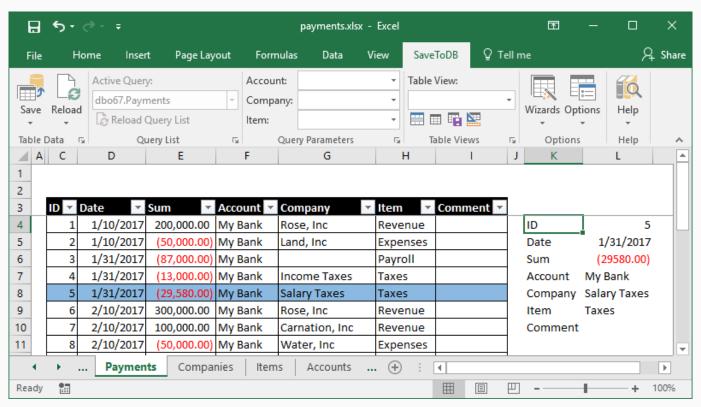
### Moving cursors using VBA

The SaveToDB add-in has special methods to move cursors: MoveFirst, MoveNext, MovePrevious, and MoveLast

```
' Moves the table cursor to the first record
Sub MoveToFirst()
   Dim addIn As Object
    Set addIn = GetAddInAndCheck()
   If addIn Is Nothing Then Exit Sub
   Call addIn.MoveFirst
End Sub
' Moves the table cursor to the previous record
Sub MoveToPrev()
    Dim addIn As Object
    Set addIn = GetAddInAndCheck()
   If addIn Is Nothing Then Exit Sub
   Call addIn.MovePrevious
End Sub
' Moves the table cursor to the next record
Sub MoveToNext()
   Dim addIn As Object
    Set addIn = GetAddInAndCheck()
   If addIn Is Nothing Then Exit Sub
   Call addIn.MoveNext
End Sub
' Moves the table cursor to the last record
Sub MoveToLast()
   Dim addIn As Object
    Set addIn = GetAddInAndCheck()
    If addIn Is Nothing Then Exit Sub
   Call addIn.MoveLast
End Sub
```

## Chapter 16. Form Fields

Let's run the same wizard, click Add Form Fields and select cell K4:



The add-in inserts form fields and updates them when a user changes the active row.

Moreover, you may use such fields to edit table row values.

#### Adding form fields using VBA

You may add and remove form fields using the AddFormFields and RemoveFormFields methods:

```
' Adding form fields, Chapter 16

Sub Chapter16_1_AddFormFields()

...

' Call addIn.RemoveFormFields(lo)
Call addIn.AddFormFields(lo, Range("K4"))

End Sub
```

As the form fields are regular Excel cells with special names, you may also use Worksheet.Names.Add and Workbook.Names.Add VBA methods to create named cells with a worksheet or workbook scope.

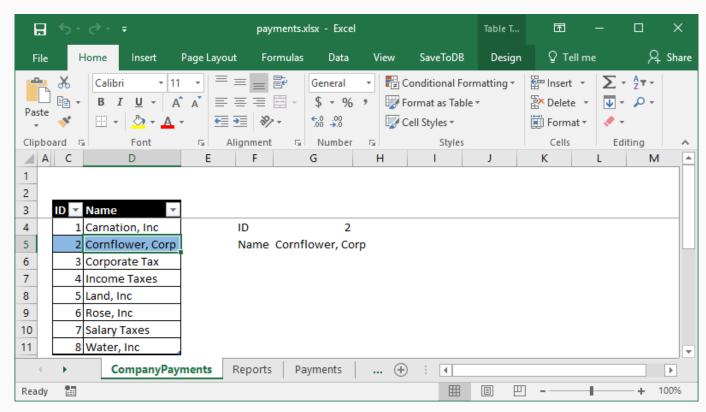
Form fields have the following name format: field <ListObjectName> <ColumnName>

The SaveToDB add-in tracks ListObject name changes and changes the field names automatically.

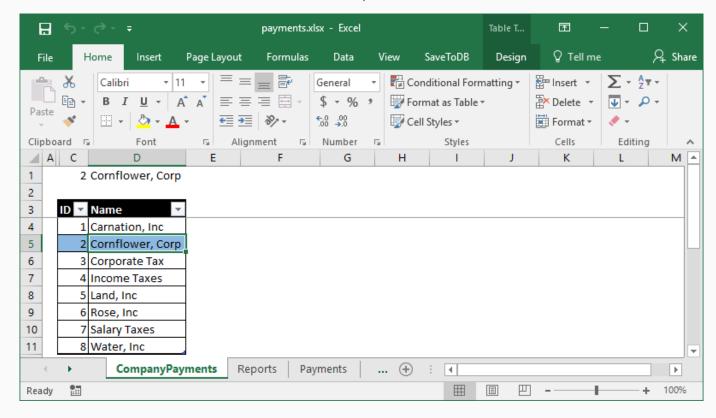
## Chapter 17. Master-Details

Let's create a new worksheet, rename it to CompanyPayments, and connect to the Company table at cell B3.

Then let's add a cursor and add form fields at cell F4.

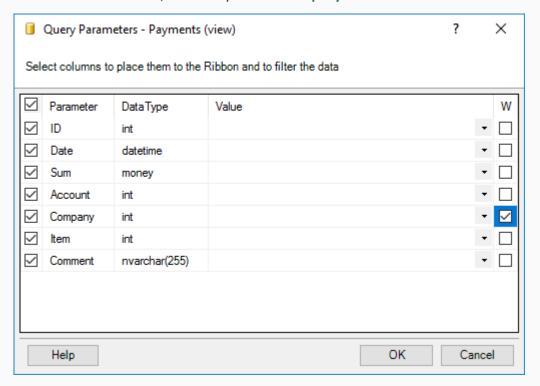


Then let's move the ID and Name fields to cells C1 and D1, and remove labels in column F:

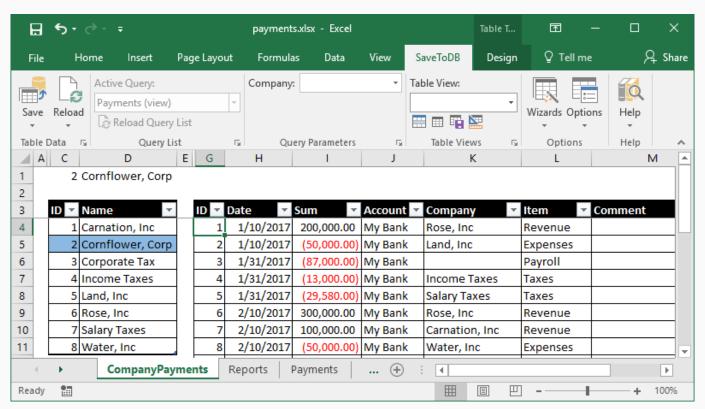


Now, select cell F3 (outside of the active table) and connect to the viewPayments view.

In the connection wizard, select only the one **CompanyID** field in the **WHERE** column:



and insert the table at cell F3:



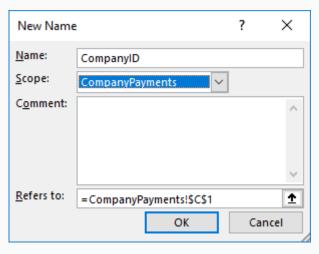
We see that the viewPayment view has only one parameter at the ribbon, Company.

Also, we see the selected company ID in cell C1 that may be used as a parameter. Let's link them.

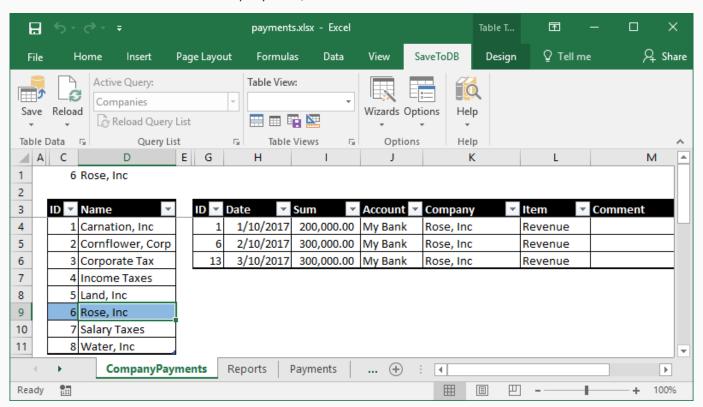
Select cell C1 and click the **Define Name** button in the **Defined Names** group on the **Formulas** tab:



Specify the cell name as the parameter name, **CompanyID**, and select the worksheet name, **CompanyPayments**, in the **Scope** field:



Click **OK**. Then select a row in the Company table, and voilà!



Here is a list of events that implement this behavior:

- 1. A user selects another row in a master table.
- 2. The add-in updates form fields (cell C1 with ID).
- 3. The add-in changes parameters of the details view using named cell values (cell C1 as CompanyID).
- 4. The add-in reloads the details with new parameter values.

You may build more complex forms using the same technique.

For example, the Northwind example contains the following tables with master-detail relations:

- 1. Customer first char index (A-Z)
- 2. Customers
- 3. Customer orders
- 4. Orders

#### Creating master-details using VBA

The following macro executes the steps described in the chapter:

```
' Creating master-details, Chapter 17
Sub Chapter17_1_CreateCompanyPaymentsWorksheet()
   Dim addIn As Object
   Set addIn = GetAddInAndCheck()
   If addIn Is Nothing Then Exit Sub
   Dim wb As Workbook
   Set wb = ActiveWorkbook
   Call addIn.InsertAddInSheets(wb)
   Dim connString As String
   connString = GetConnectionString()
   Dim ws As Worksheet
   Set ws = GetOrCreateWorksheet(wb, "CompanyPayments")
   ws.Select
   If ws.ListObjects.Count > 0 Then
       MsgBox "Worksheet CompanyPayments already has tables"
        Exit Sub
   End If
   Call InsertConnectedListObject(addIn,
           Range("B3"), connString, "dbo67.Companies", "TABLE", True)
   If ws.ListObjects.Count < 1 Then Exit Sub</pre>
   Call InsertConnectedListObject(addIn,
           Range("F3"), connString, "dbo67.viewPayments", "VIEW", True)
   If ws.ListObjects.Count < 2 Then Exit Sub</pre>
```

```
Dim lo As ListObject
    Set lo = ws.ListObjects(1)
    Call addIn.AddTableCursor(lo)
    Dim fieldIdName As String
    fieldIdName = addIn.GetFormFieldCellName(lo, "ID")
    Dim fieldNameName As String
    fieldNameName = addIn.GetFormFieldCellName(lo, "Name")
    ws.Names.Add Name:=fieldIdName, RefersTo:=Range("C1")
    ws.Names.Add Name:=fieldNameName, RefersTo:=Range("D1")
    Set lo = ws.ListObjects(2)
    addIn.IsRibbonField(lo, "AccountID") = False
addIn.IsRibbonField(lo, "CompanyID") = True
addIn.IsRibbonField(lo, "ItemID") = False
    ' Set the initial value using ID from cell C1 (or using Name from cell D1)
    addIn.ParameterValue(lo, "CompanyID") = Range("C1").Value
    'addIn.ParameterRibbonValue(lo, "CompanyID") = Range("D1").Value
    ws.Names.Add Name:="CompanyID", RefersTo:=Range("C1")
    ' Cell C1 must have two names: field CompanyPayments Table1 ID and CompanyID
    ws.Range("D4").Select
End Sub
```

It creates a new worksheet and inserts two tables, dbo67.Payments and dbo67.viewPayments.

Then it adds a table cursor.

The code uses tricks to create the ID and Name form fields in the specified cells, C1 and D1.

You may use the **GetFormFieldCellName** function to get form field names and the VBA Worksheet.Names.Add and Workbook.Names.Add methods to create form field cells.

Then the code configures ribbon fields for the detail table. It leaves the CompanyID field only.

Then the code creates the CompanyID name for the form field in cell C1.

So, when a user changes a row in a parent table, the add-in updates the form fields, then updates parameters and reloads detail tables with new parameter values.

# Chapter 18. Detail Windows and Task Panes

We may use another technique to show details in windows and task panes using the **SelectionChange** handlers.

#### **Detail Windows**

Let's switch to the payments-configuration workbook, select the EventHandlers worksheet and add the data:

ID 🔻	<b>TABLE</b> $\forall$	TABLE_NAME 🔻	COLUMN EVENT_NAME	HANI 🔻	HANDLER_NAME V	HANDLER ~	HANDLER_CODE ~	TARGET_WORKSHEET
35	dbo67	viewPayments	SelectionChange	dbo67	Company Payments	CODE	SELECT p.[Date]	
36	dbo67	viewPayments	SelectionChange	dbo67	Item Payments	CODE	SELECT p.[Date]	

The **HANDLER\_CODE** of the **Company Payments** handler contains the code:

```
SELECT
   p.[Date]
   , p.[Sum]
   , c.Name AS Company
   , i.Name AS Item
   , p.Comment
FROM
   dbo67.Payments p
   LEFT OUTER JOIN dbo67.Items i ON i.ID = p.ItemID
   INNER JOIN dbo67.Companies c ON c.ID = p.CompanyID
WHERE
   p.CompanyID = @CompanyID
```

The **HANDLER\_CODE** of the **Item Payments** handler contains the code:

```
SELECT
    p.[Date]
    , p.[Sum]
    , c.Name AS Company
    , i.Name AS Item
    , p.Comment
FROM
    dbo67.Payments p
    INNER JOIN dbo67.Items i ON i.ID = p.ItemID
    LEFT OUTER JOIN dbo67.Companies c ON c.ID = p.CompanyID
WHERE
    p.ItemID = @ItemID
```

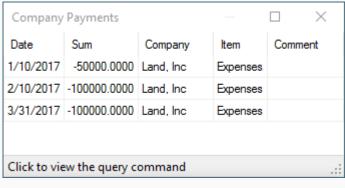
As we may suppose, the add-in executes the specified code on the **SelectionChange** event.

The handlers use @CompanyID and @ItemID parameters accordingly to filter output data.

Let's switch to the payments workbook, select the Reports worksheet, and select the Payments (view) view.

If the view is already activated, click **Reload**, **Reload Data and Configuration**.

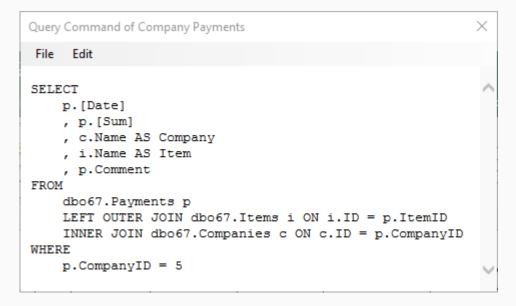
Now, when we change a row, the add-in launches windows like this:





We may select another row. Windows will stay on top and show related information.

You may click on the window status line to see the executed SQL command like this:



In this example, we have used the direct SQL codes stored in the **EventHandlers** table.

This is useful when you do not want to modify your database.

Otherwise, you may use stored procedures with parameters.

#### **Cell Editor**

The SaveToDB add-in adds great support for editing codes in cells.

When the cell contains a multiline value, the add-in launches the **Cell Editor** like this:

```
Cell Editor
                                                          ×
💹 | 🐰 🛅 🖺 | " | Select All | Consolas
                                     11
SELECT
    p.[Date]
    , p.[Sum]
    , c.Name AS Company
    , i.Name AS Item
    , p.Comment
FROM
    dbo67.Payments p
    LEFT OUTER JOIN dbo67.Items i ON i.ID = p.ItemID
    INNER JOIN dbo67.Companies c ON c.ID = p.CompanyID
WHERE
    p.CompanyID = @CompanyID
```

You may edit the text in the editor and click the **Save** button to update the underlying cell.

You may turn on/off the editor using the Options, Show Cell Editor Task Pane option.

#### **Task Panes**

Use of detail windows is a good solution when you need windows that are always visible.

If you need context windows, you may use task panes.

Let's add the following handlers to the **EventHandlers** table for the **uspPayments** procedure:

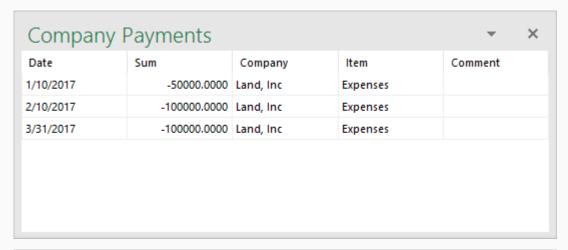
ID 🔻	TABLE 🔻	TABLE_NAME *	COLUMN V EVENT_NAME V	HANI▼	HANDLER_NAME -	HANDLER -	HANDLER_CODE ~	TARGET_WORKSHEET
35	dbo67	viewPayments	SelectionChange	dbo67	Company Payments	CODE	SELECT p.[Date]	
36	dbo67	viewPayments	SelectionChange	dbo67	Item Payments	CODE	SELECT p.[Date]	
37	dbo67	uspPayments	SelectionChange	dbo67	Company Payments	CODE	SELECT p.[Date]	_TaskPane
38	dbo67	uspPayments	SelectionChange	dbo67	Item Payments	CODE	SELECT p.[Date]	_TaskPane

I have copied the view rows, changed the view name, and added **\_TaskPane** to the **TARGET\_WORKSHEET** field.

The **HANDLER\_CODE** codes remain the same.

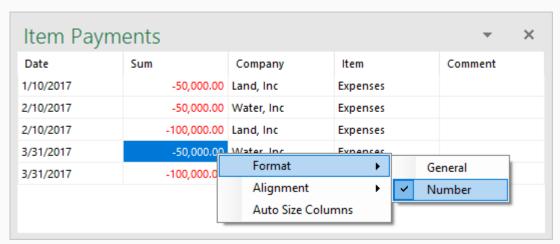
Let's switch to the payments workbook, and change the query to the Payments (sp) procedure.

When we change a row, the add-in shows task panes like this:





Users may customize column formats:



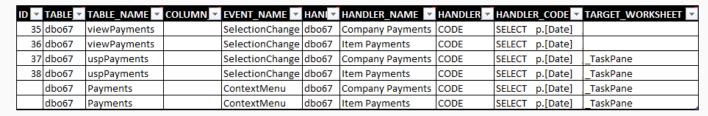
Also, users may dock task panes and turn them on/off using the **Options**, **Show Task Panes** option.

## Chapter 19. Context Menus

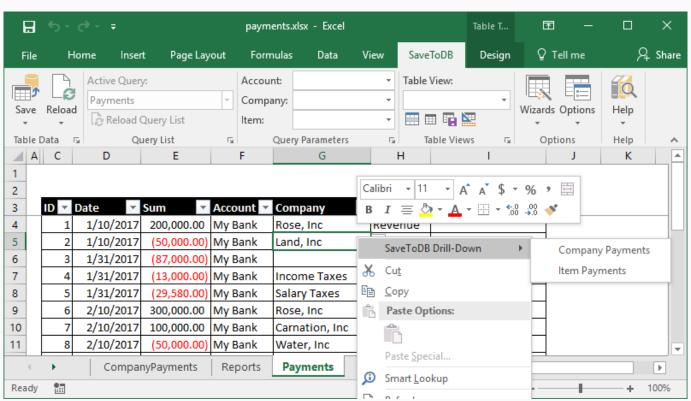
Detail windows and task panes discussed above are shown in the **SelectionChange** event.

We may add such queries and much more to the context menu.

Let's add the following configuration to the EventHandlers table (two lines at the bottom):



Then let's switch to the **payments** workbook, select the **Payments** worksheet, click **Reload**, **Reload Data and Configuration**, and right click on a row:



When we click on the Company Payments item, the add-in shows the task pane like this:



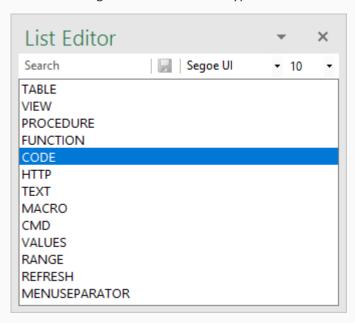
As we learned above, the **SelectionChange** and **ContextMenu** handlers are nearly the same.

They get parameters from the columns of the active row. They differ by the **EVENT\_NAME** configuration.

However, the **ContextMenu** handlers support additional configuration parameters like **MENU\_ORDER** and **EDIT\_PARAMETERS**:

EVENT_NAME 🔻	HANI▼	HANDLER_NAME 🔻	<b>HANDLER</b> ▼	HANDLE	R_CODE ~	TARGET_WORKSHEET	MENU_ORDER 🔻	EDIT_PARAMETERS 🔻
ContextMenu	dbo67	Company Payments	CODE	SELECT	p.[Date]	_TaskPane		
ContextMenu	dbo67	Item Payments	CODE	SELECT	p.[Date]	_TaskPane		

and allow using much more handler types:



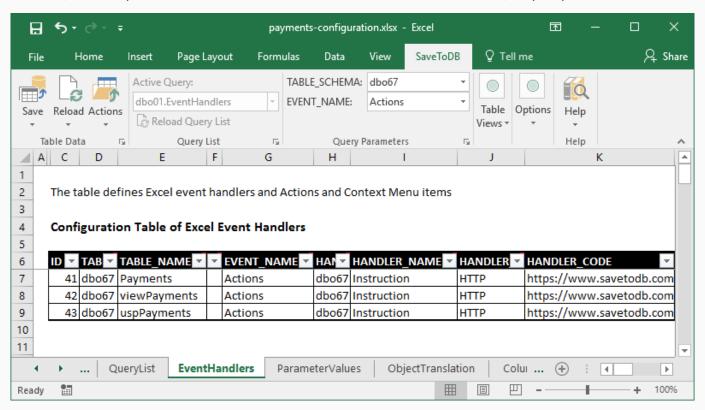
You may configure executing stored procedures, SQL codes, macros, CMD commands, opening URLs.

Use the **MenuSeparator** type to separate menu items.

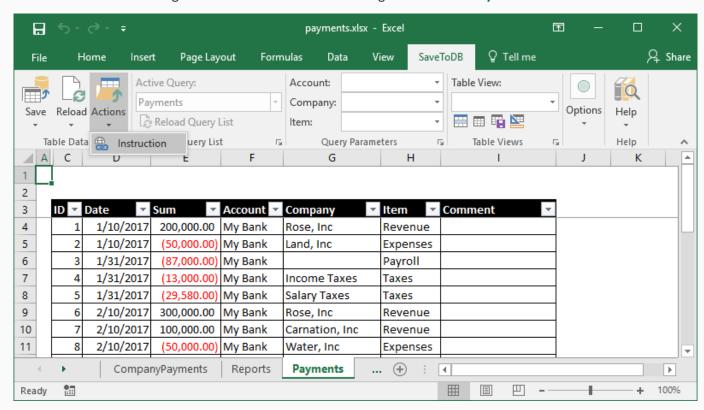
## Chapter 20. Actions Menus

The action menus are located at the ribbon and have nearly the same features as context menus.

However, users may execute actions when the active cell is outside of the table. So, they may have no context.



Let's add the Actions configuration and reload data and configuration at the Payments worksheet:



## Chapter 21. Creating and Removing Applications

Using the add-in, you work with high-level methods while the add-in loads configurations from a database.

So, you may create applications completely using VBA from empty workbooks like in this book.

```
' Creates the application worksheets, Chapter 21
Sub Chapter21_1_CreateApplication()

Call Chapter02_2_CreateMasterTableEditors()
Call Chapter05_1_CreatePaymentsWorksheet()
Call Chapter06_1_SetPaymentsWhereParameters()
Call Chapter07_1_SetLanguage()
Call Chapter09_1_CreateTableViews()
Call Chapter09_1_CreateReportsWorksheet()
Call Chapter11_1_CreateReportsWorksheet()
Call Chapter13_1_ChangeQueryObject()
Call Chapter14_1_SaveChanges()
Call Chapter15_1_AddCursors()
Call Chapter15_2_AddCursorButtons()
Call Chapter16_1_AddFormFields()
Call Chapter17_1_CreateCompanyPaymentsWorksheet()
End Sub
```

When you remove an application, remove the add-in data sheets using the last three methods:

```
' Removes the application worksheets, Chapter 21
Sub Chapter21_2_RemoveApplication()
   If Not MsgBox("Are you sure to delete all worksheets?", vbYesNo) = vbYes Then Exit Sub
   Dim wb As Workbook
   Set wb = ActiveWorkbook
   Call DeleteWorksheet(wb, "CompanyPayments")
   Call DeleteWorksheet(wb, "Reports")
   Call DeleteWorksheet(wb, "Payments")
   Call DeleteWorksheet(wb, "Accounts")
   Call DeleteWorksheet(wb, "Companies")
   Call DeleteWorksheet(wb, "Items")
   Dim addIn As Object
   Set addIn = GetAddInAndCheck()
   If addIn Is Nothing Then Exit Sub
   Call addIn.DeleteAllTableViews(wb)
   Call addIn.DeleteAllTableCursors(wb)
   Call addIn.DeleteAddInSheets(wb)
    ' The workbook must contain the start sheet only
End Sub
```

## Conclusion

We have created a complete client application using Microsoft Excel.

Here are ten steps that we made:

- 1. Connect to tables, views, and stored procedures
- 2. Configure validation lists
- 3. Configure ribbon parameters
- 4. Translate field, parameter, and object names
- 5. Configure formats, formulas, and table views
- 6. Configure saving changes
- 7. Add cursors and form fields
- 8. Create master-detail forms
- 9. Configure detail windows and task panes
- 10. Configure context and action menus

The most of the features are configured in a database, using configuration tables and SQL codes.

This is a big advantage as this solves all deployment challenges. Users just reload the configuration.

As a VBA developer, you may use these features from-the-box and add only specific features.

I hope it helps you build advanced applications with less time and efforts getting satisfied users and more money.

You may download the SaveToDB add-in at www.savetodb.com.

All features described in this book are available in the free SaveToDB Express edition.

I would appreciate your feedback. Feel free to contact me at 10-vba-steps@savetodb.com.

Happy coding!

Sergey Vaselenko

## About the Author



My name is Sergey Vaselenko.

I am from Russia, Moscow.

My passion is creating software.

I am a founder and CEO of Gartle Technology Corporation and a leading developer of the SaveToDB add-in.

You are welcome to contact me at

www.facebook.com/sergey.vaselenko

www.linkedin.com/in/vaselenko/

## Appendix 1. Database Source Code

You may download the source code at

https://www.savetodb.com/download.php?file=10-steps-for-vba-developers.zip

You may download the SaveToDB add-in at www.savetodb.com.

To use all described features, install version SaveToDB 7.2 or higher.

#### **Master Tables**

```
CREATE TABLE [dbo67].[Accounts] (
      [ID] int IDENTITY(1,1) NOT NULL
    , [Name] nvarchar(50) NOT NULL
    , CONSTRAINT [PK Accounts dbo67] PRIMARY KEY ([ID])
    , CONSTRAINT [IX Accounts dbo67] UNIQUE ([Name])
)
GO
CREATE TABLE [dbo67].[Companies] (
      [ID] int IDENTITY(1,1) NOT NULL
    , [Name] nvarchar(50) NOT NULL
    , CONSTRAINT [PK_Companies_dbo67] PRIMARY KEY ([ID])
    , CONSTRAINT [IX Companies dbo67] UNIQUE ([Name])
G0
CREATE TABLE [dbo67].[Items] (
      [ID] int IDENTITY(1,1) NOT NULL
    , [Name] nvarchar(50) NOT NULL
    , CONSTRAINT [PK_Items_dbo67] PRIMARY KEY ([ID])
    , CONSTRAINT [IX_Items_dbo67] UNIQUE ([Name])
)
GO
```

Add UNIQUE constraints to your tables to avoid name doubles.

### **Table dbo67.Payments**

```
CREATE TABLE [dbo67].[Payments] (
      [ID] int IDENTITY(1,1) NOT NULL
    , [Date] datetime NULL
    , [Sum] money NULL
    , [AccountID] int NULL
    , [CompanyID] int NULL
    , [ItemID] int NULL
    , [Comment] nvarchar(255) NULL
    , CONSTRAINT [PK Payments dbo67] PRIMARY KEY ([ID])
)
GO
ALTER TABLE [dbo67]. [Payments] ADD CONSTRAINT [FK_Payments_Accounts_dbo67]
FOREIGN KEY ([AccountID]) REFERENCES [dbo67].[Accounts] ([ID]) ON UPDATE CASCADE
GO
ALTER TABLE [dbo67].[Payments] ADD CONSTRAINT [FK_Payments_Companies_dbo67]
FOREIGN KEY ([CompanyID]) REFERENCES [dbo67].[Companies] ([ID]) ON UPDATE
CASCADE
GO
ALTER TABLE [dbo67].[Payments] ADD CONSTRAINT [FK Payments Items dbo67]
FOREIGN KEY ([ItemID]) REFERENCES [dbo67].[Items] ([ID]) ON UPDATE CASCADE
GO
```

#### View dbo67.viewPayments

```
CREATE VIEW [dbo67].[viewPayments]

AS

SELECT

p.ID

p.[Date]

p.[Sum]

p.AccountID

p.CompanyID

p.ItemID

p.Comment

FROM

dbo67.Payments p
```

The view selects data from a single table that makes it updateable.

You may use INSERT, UPDATE, and DELETE statements for such views directly. The add-in uses this feature.

### **Stored Procedure dbo67.uspPayments**

```
CREATE PROCEDURE [dbo67].[uspPayments]
    @AccountID int = NULL
    , @CompanyID int = NULL
    , @ItemID int = NULL
AS
BEGIN
SET NOCOUNT ON
SELECT
    p.ID
    , p.[Date]
    , p.[Sum]
    , p.AccountID
    , p.CompanyID
    , p.ItemID
    , p.Comment
FROM
    dbo67.Payments p
WHERE
    COALESCE(@AccountID, p.AccountID, 0) = COALESCE(p.AccountID, 0)
    AND COALESCE (@CompanyID, p.CompanyID, 0) = COALESCE (p.CompanyID, 0)
    AND COALESCE (@ItemID, p.ItemID, 0) = COALESCE (p.ItemID, 0)
END
G0
```

The stored procedure has parameters used to filter source table data. The NULL value means "all values."

Use **SET NOCOUNT ON** as the first command. Otherwise, Excel cannot load data from the procedure.