Getting Started with ExcelMVC

Just like Silverlight or WPF (Windows Presentation Foundation), ExcelMVC facilitates a clear separation between your application's business objects (Models), its user interfaces (Views) and its controllers (View Models).

With ExcelMVC, you declare your views in Excel (as opposed to in XAML with Silverlight or WPF) and bind to them their models implemented in any .NET languages. Your views are Excel worksheets, but your models are free of Excel specifics.

At the end of this tutorial, you will see that developing Excel applications with ExcelMVC is much simpler than developing their WPF or Silverlight equivalent ones, as you can rely on Excel doing the heavy lifting for most (if not all) of your applications' UI functions.

Just imagine how difficult it would be for a non-Excel application to replicate just some of the Excel UI features, e.g. formatting, charting, printing, import /export etc.

With ExcelMVC, an application can take advantage of Excel's unbelievably rich UI and in the meantime clearly separate the application's business layer from its UI layer, making it testable with testing tools (e.g. MSTest and NUnit).

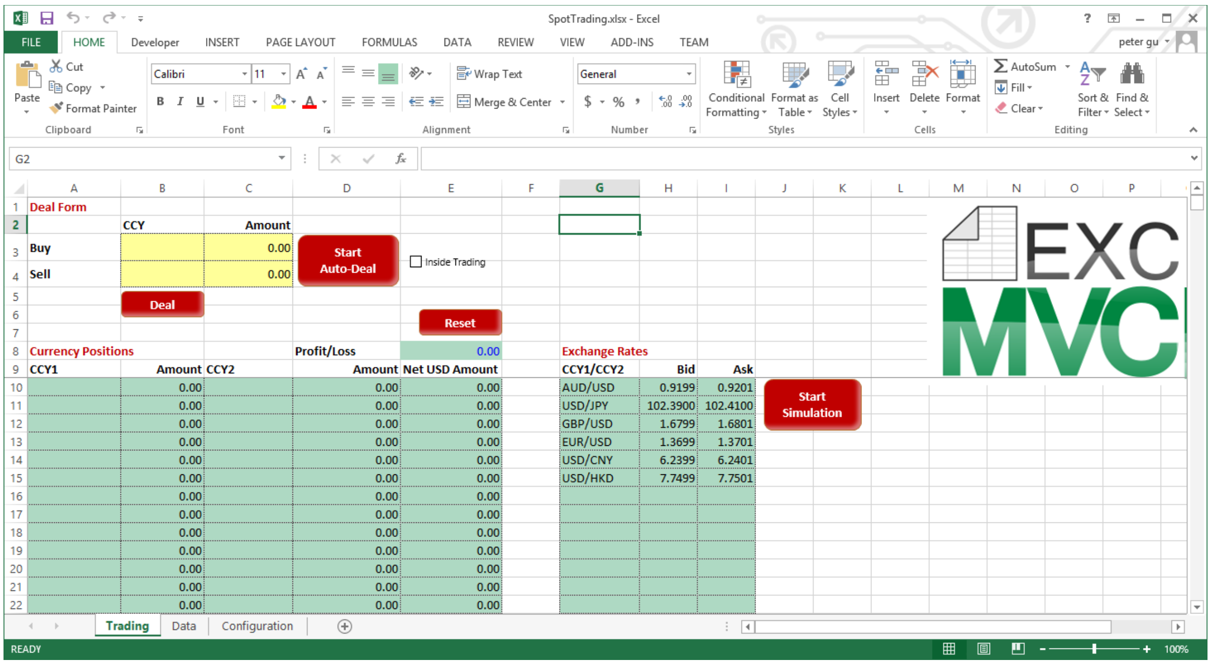
So without further ado, let's get started.

Downloading ExcelMVC sample applications

First let's see what an ExcelMVC application looks like. Please follow the steps below to run the FX spot trading application.

1. Click here to download the latest ExcelMVC release and unzip the release to a folder on a local disk.
2. Go to the "Samples\Trading" folder, unblock all executable files (*.dll, *.xll and *.cmd). Right click on each file and select Properties to unblock the file.
3. Double click on the "Run.cmd" file to start the sample application. You will see an Excel screen shown below.
4. Click on "Start Auto-Deal" and "Start Simulation", and you are on your way of making (or losing) loads of money from the foreign exchange spot market.

5. Navigate to other tabs to get a basic idea on how ExcelMVC glues every things together. Press Alt + F11 to see there is absolutely no VB code behind.



	A	B	C	D	E	F	G	H	I
1	Deal Form								
2		CCY	Amount						
3	Buy		0.00						
4	Sell		0.00						
5									
6									
7									
8	Currency Positions			Profit/Loss	0.00		Exchange Rates		
9	CCY1	Amount	CCY2	Amount	Net USD Amount		CCY1/CCY2	Bid	Ask
10		0.00		0.00	0.00		AUD/USD	0.9199	0.9201
11		0.00		0.00	0.00		USD/JPY	102.3900	102.4100
12		0.00		0.00	0.00		GBP/USD	1.6799	1.6801
13		0.00		0.00	0.00		EUR/USD	1.3699	1.3701
14		0.00		0.00	0.00		USD/CNY	6.2399	6.2401
15		0.00		0.00	0.00		USD/HKD	7.7499	7.7501
16		0.00		0.00	0.00				
17		0.00		0.00	0.00				
18		0.00		0.00	0.00				
19		0.00		0.00	0.00				
20		0.00		0.00	0.00				
21		0.00		0.00	0.00				
22		0.00		0.00	0.00				

Creating an ExcelMVC application

A typical ExcelMVC application is made up by two parts: the Application's UI (Excel workbooks) and its code behind (.NET assemblies). Each part can be developed by different teams. For example expert Excel users can design the application's workbooks, while professional .NET developers can prepare and test the application code, independently if necessary. These two parts are then glued together by ExcelMVC.

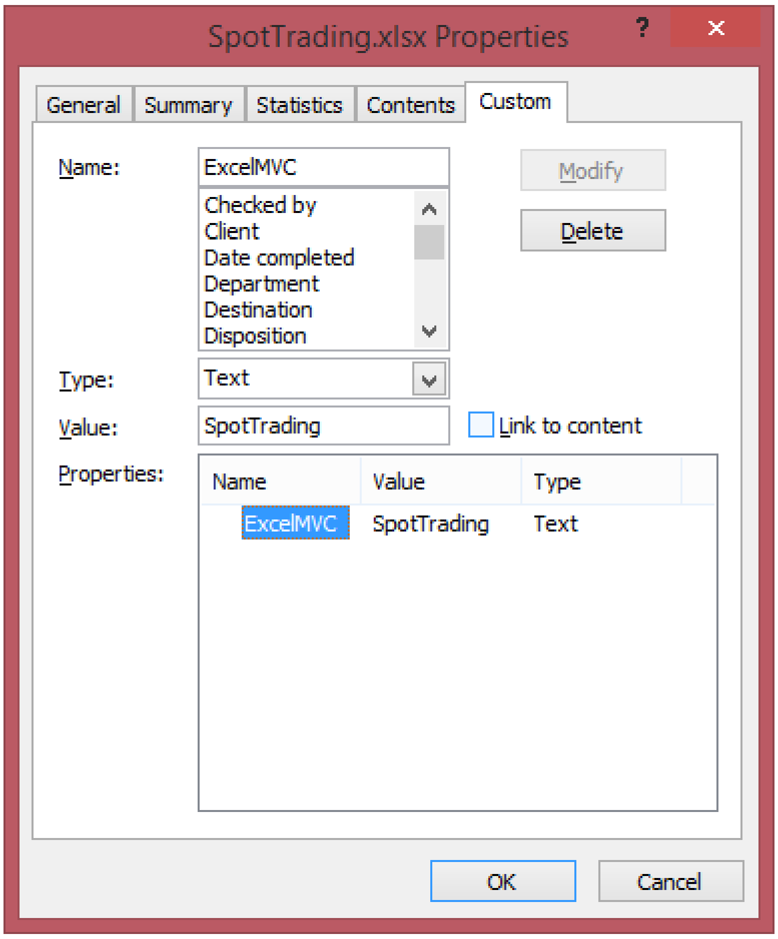
We will use the FX spot trading sample to work through the process of creating an ExcelMVC project.

Designing ExcelMVC workbooks

Identifying ExcelMVC workbooks

An ExcelMVC workbook is identified by a custom property named "ExcelMVC" (Please note ExcelMVC string comparison is case insensitive).

The picture below shows the value of this property for the FX spot trading workbook (go to File | Properties | Advanced Properties).


SpotTrading.xlsx Properties

Name: ExcelMVC

Type: Text

Value: SpotTrading

Properties:

Name	Value	Type
ExcelMVC	SpotTrading	Text

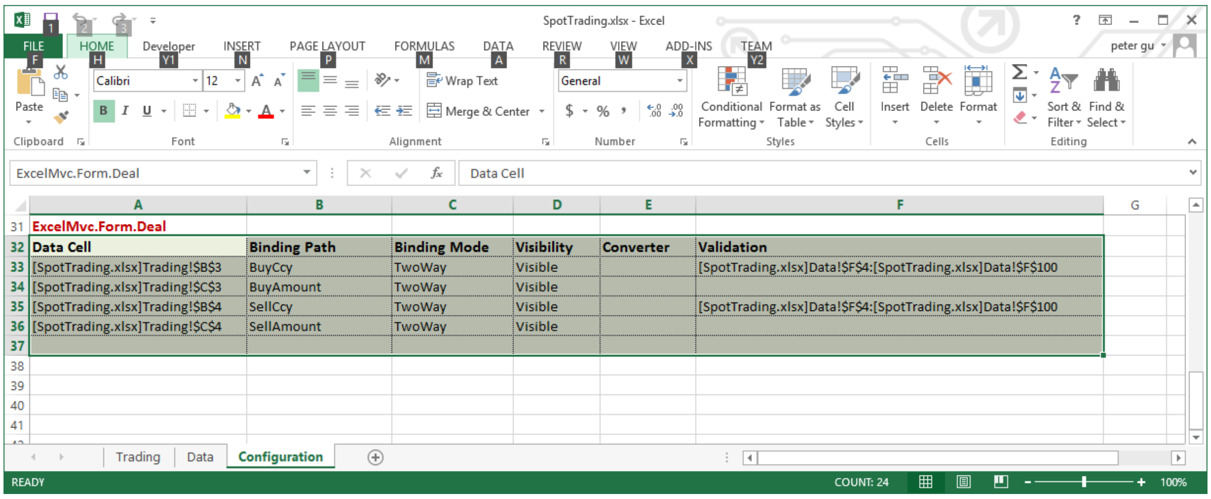
Later in this tutorial, you will see how ExcelMVC notifies your application whenever a workbook is about to be opened in an Excel session. You use this property to tell if a workbook belongs to your ExcelMVC application.

Defining ExcelMVC Forms and Tables

ExcelMVC currently supports two types of views, *Form* and *Table*. A form facilitates the binding between the properties of an object and Excel cells, while a table facilitates the binding between a collection of objects and Excel rows or columns.

Both views must be declared using Excel named ranges, which can be located anywhere in a workbook.

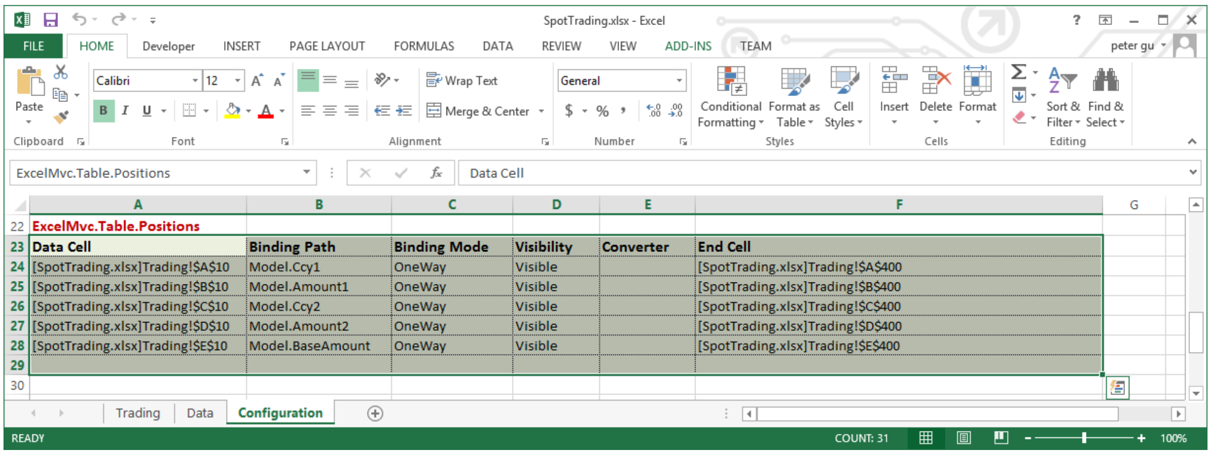
The name for an ExcelMVC form definition range must be in the form of "ExcelMVC.Form.X", where X is the actual form name. The picture below shows the "Deal" form definition in the FX spot trading workbook.



ExcelMvc.Form.Deal | Data Cell

	A	B	C	D	E	F
31	ExcelMvc.Form.Deal					
32	Data Cell	Binding Path	Binding Mode	Visibility	Converter	Validation
33	[SpotTrading.xlsx]Trading!$B$3	BuyCcy	TwoWay	Visible		[SpotTrading.xlsx]Data!$F$4:[SpotTrading.xlsx]Data!$F$100
34	[SpotTrading.xlsx]Trading!$C$3	BuyAmount	TwoWay	Visible		
35	[SpotTrading.xlsx]Trading!$B$4	SellCcy	TwoWay	Visible		[SpotTrading.xlsx]Data!$F$4:[SpotTrading.xlsx]Data!$F$100
36	[SpotTrading.xlsx]Trading!$C$4	SellAmount	TwoWay	Visible		
37						

Similarly, the name for an ExcelMVC table definition range must be in the form of "ExcelMVC.Table.Y", where Y is the actual table name. The picture below shows the "Positions" table definition in the FX spot trading workbook.



ExcelMvc.Table.Positions | Data Cell

	A	B	C	D	E	F
22	ExcelMvc.Table.Positions					
23	Data Cell	Binding Path	Binding Mode	Visibility	Converter	End Cell
24	[SpotTrading.xlsx]Trading!$A$10	Model.Ccy1	OneWay	Visible		[SpotTrading.xlsx]Trading!$A$400
25	[SpotTrading.xlsx]Trading!$B$10	Model.Amount1	OneWay	Visible		[SpotTrading.xlsx]Trading!$B$400
26	[SpotTrading.xlsx]Trading!$C$10	Model.Ccy2	OneWay	Visible		[SpotTrading.xlsx]Trading!$C$400
27	[SpotTrading.xlsx]Trading!$D$10	Model.Amount2	OneWay	Visible		[SpotTrading.xlsx]Trading!$D$400
28	[SpotTrading.xlsx]Trading!$E$10	Model.BaseAmount	OneWay	Visible		[SpotTrading.xlsx]Trading!$E$400
29						

Each data row in a Form or Table definition range specifies a binding settings between a property of an object and an Excel cell.

The heading row in a definition range is what ExcelMVC uses to parse binding settings. The ordering of the headings is insignificant, but the actual heading text is significant. The table below lists headings required by ExcelMVC.

Heading	**Description**
Data Cell/Start Cell	The address of the cell which the property specified by the Binding Path is bound to. Use Excel function CELL function, e.g.

Heading	Description
	“=CELL("address", Trading!A10)” to acquire an address string.
Data Path	The path to the property to be bound to the cell specified by Data Cell
Binding Mode	Determines how the binding is done between a view and its model. The following modes are supported: **OneWay** – Properties on a model are displayed on the view to which the model is bound to. Changes made on a view is not updated to its model. This mode is primary for static views. **OneWayToSource** – Values on a view are copied to the model to which the view is bound to. Changes made on a model is not displayed on its view. This mode is primary for static models with their properties sourced for Excel. **TwoWay** – Properties on a model and values on its view are exchanged in both directions, model to view and view to model.
Visibility (Optional, default is True)	True or False, indicates the visibility of a table column (portrait) or table row (landscape). This setting does not apply for forms. See below for supported table orientations.
Converter (Optional)	Specifies the converter (an instance of a class derived from System.Wndows.Data.IValueConverter). The hosting assemblies of converters must be in the application base path.
Validation (Optional)	Specifies a range to be used as the Excel validation list of the cell. Use Excel function CELL and CONCATENATE to specify a validation range string, e.g. =CONCATENATE(CELL("address", Data!F4), ":", CELL("address",Data!F100))
End Cell (optional)	Specifies the last cell to which a table should be bound. If not specified, a table will be bound to as many rows as the

Heading	Description
	number of objects available in its view code.

Deciding ExcelMVC Table Orientations

ExcelMVC tables can be defined in two orientations, Portrait and Landscape.

A portrait table binds each object in its view model to an Excel row. To define a portrait table, simply specify the Data Path of each binding on a single Excel row.

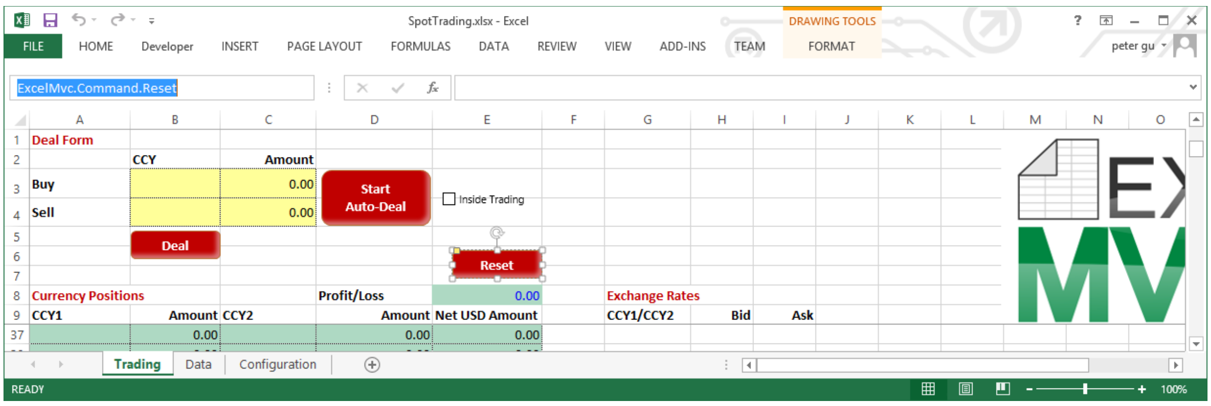
A landscape table binds each object in its view model to an Excel column. To define a landscape table, simply specify the Data Path of each binding on a single Excel column.

Defining ExcelMVC Commands

ExcelMVC supports the following Excel form controls as commands.

- Button
- Check Box
- Option Button
- Combo Box
- List Box
- Shape
- Spin Button

To define a form control as an Excel command, all you have to do is to name it in the form of "ExcelMVC.Commad.Z", where Z is the actual command name. The picture below shows how the "Reset" button is named.



ExcelMvc.Command.Reset

	A	B	C	D	E	F	G	H	I
1	Deal Form								
2		CCY	Amount						
3	Buy		0.00						
4	Sell		0.00						
8	Currency Positions			Profit/Loss	0.00		Exchange Rates		
9	CCY1	Amount	CCY2	Amount	Net USD Amount		CCY1/CCY2	Bid	Ask
37		0.00		0.00	0.00				

ExcelMVC currently does not support ActiveX controls.

Setting up an ExcelMVC .NET Solution

An ExcelMVC application typically requires four groups of projects:

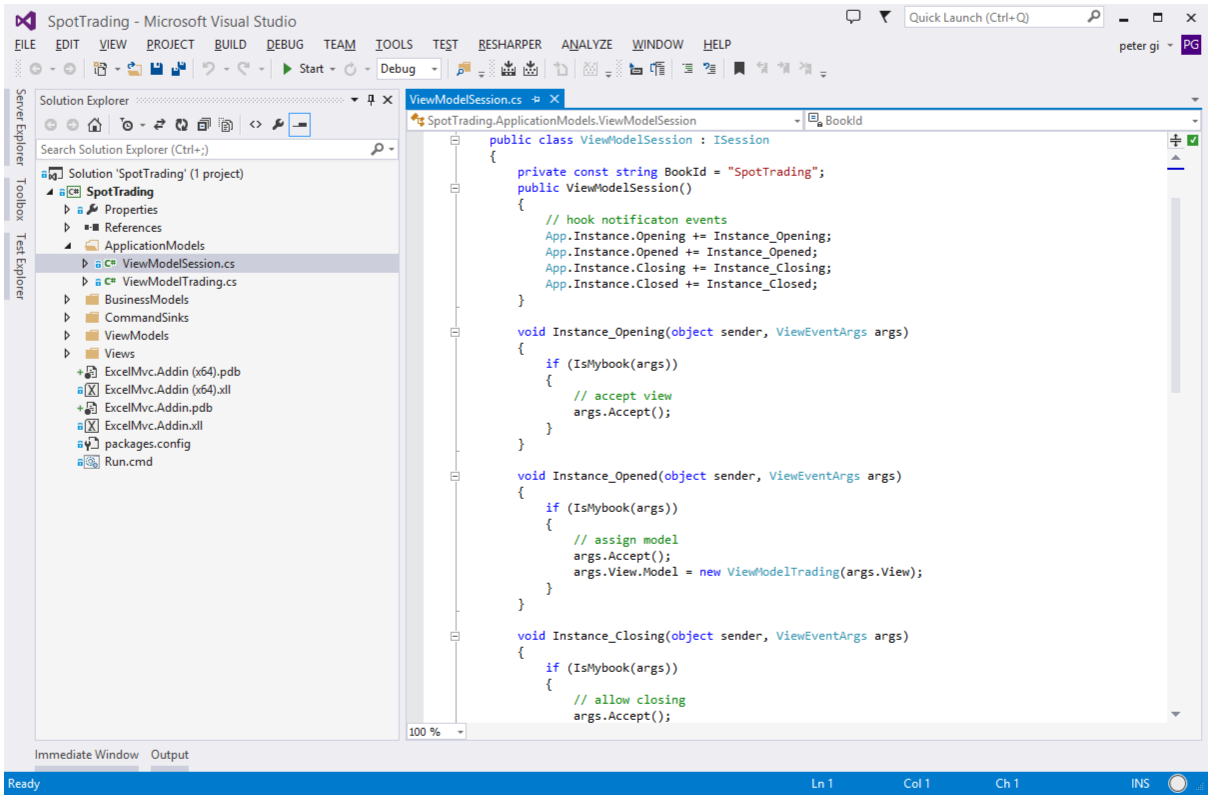
1. Business model assemblies, with each containing classes which are free of UI or binding details.
2. View model assemblies, with each containing classes derived from System.ComponentModel.INotifyPropertyChanged (for binding forms, optional), or System.Collections.Specialized.IEnumerable (and optionally from System.Collections.Specialized.INotifyCollectionChanged) (for binding tables), or System.Windows.Input.ICommand (for binding commands).
3. Application session assemblies (referencing ExcelMVC), with each containing classes derived from ExcelMvc.Runtime.ISession (for binding ExcelMVC views).
4. View assemblies, with each containing Excel workbooks and any Window forms (WPF or WinForm) required by the application. If your application only uses ExcelMVC views, then there is no need to place your workbooks in a separate assembly.

The steps below show how a typical ExcelMVC application is created.

Start Visual Studio 2013 now and open the FX spot trading solution (located under “Samples\Trading”) and check the solution structure against the steps listed below. (Note for the sake of simplicity and brevity, we used different namespaces instead different assemblies for each group of assemblies listed under 1 to step 3.)

1. Create application’s business classes in a C# class library project, referencing any data access assemblies, web service client assemblies, or other business object assemblies.
2. Create application’s view model classes in a C# class library, referencing System.Data, System.Windows.
3. Create application’s session classes in a C# class library project, referencing ExcelMVC assembly. Add an App.config file to the project if your application reads settings from the .NET application configuration file. To add ExcelMVC to the project, the easiest way is to use NuGet Package Manager to install and update ExcelMVC. You will need “NuGet

Package Manager for Visual Studio 2013" installed (through "Tools | Extensions and Updates"). Make sure you have "nuget.org" as one of the package sources (see settings under Tools | NuGet Package Manager | Package Manager Settings).



```
public class ViewModelSession : ISession
{
    private const string BookId = "SpotTrading";
    public ViewModelSession()
    {
        // hook notificaton events
        App.Instance.Opening += Instance_Opening;
        App.Instance.Opened += Instance_Opened;
        App.Instance.Closing += Instance_Closing;
        App.Instance.Closed += Instance_Closed;
    }

    void Instance_Opening(object sender, ViewEventArgs args)
    {
        if (IsMybook(args))
        {
            // accept view
            args.Accept();
        }
    }

    void Instance_Opened(object sender, ViewEventArgs args)
    {
        if (IsMybook(args))
        {
            // assign model
            args.Accept();
            args.View.Model = new ViewModelTrading(args.View);
        }
    }

    void Instance_Closing(object sender, ViewEventArgs args)
    {
        if (IsMybook(args))
        {
            // allow closing
            args.Accept();
```

4. Add your workbooks and any Windows forms to a C# class library, if required, referencing to System.Windows.Forms, PresentationCore and PresentationFramework. Again if you are using ExcelMVC views only, then you don't need this assembly.
5. Now, try to create your own ExcelMVC solution by following the above steps. Other than referencing the ExcelMVC assembly in your application's session assembly, there is really no difference between setting up an ordinary .NET solution and setting up an ExcelMVC solution.

Creating an ExcelMVC Session

ExcelMVC binds application's views to their models through its session object derived from ExcelMVC.Runtime.ISession. An application's session object is discovered and created by

ExcelMVC automatically when the application is started with ExcelMVC.Addin.xll (located in the application's directory).

The ViewModelSession session class below is from the FX spot trading sample. The code comments pretty much describes how a typical ExcelMVC session object handles the ExcelMVC book view notification events. The book id "SpotTrading" must be equal to the value of the custom property "ExcelMVC", as described in section Identifying ExcelMVC workbooks.

```
namespace SpotTrading.ApplicationModels
{
    using ExcelMvc.Extensions;
    using ExcelMvc.Runtime;
    using ExcelMvc.Views;

    public class ViewModelSession : ISession
    {
        private const string BookId = "SpotTrading";
        public ViewModelSession()
        {
            // hook notificaton events
            App.Instance.Opening += Instance_Opening;
            App.Instance.Opened += Instance_Opened;
            App.Instance.Closing += Instance_Closing;
            App.Instance.Closed += Instance_Closed;
        }

        void Instance_Opening(object sender, ViewEventArgs args)
        {
            if (IsMybook(args))
            {
                // accept view
                args.Accept();
            }
        }

        void Instance_Opened(object sender, ViewEventArgs args)
        {
            if (IsMybook(args))
            {
                // assign model
                args.Accept();
                args.View.Model = new ViewModelTrading(args.View);
            }
        }

        void Instance_Closing(object sender, ViewEventArgs args)
        {
            if (IsMybook(args))
            {
                // allow closing
                args.Accept();
            }
        }

        void Instance_Closed(object sender, ViewEventArgs args)
        {
            if (IsMybook(args))
            {
                // detach model
                args.View.Model = null;
            }
        }

        private bool IsMybook(ViewEventArgs args)
        {
            return args.View.Id.CompareOrdinalIgnoreCase(BookId) == 0;
        }

        public void Dispose()
        {
        }
    }
}
```

In this sample, the view model for the book identified as the "`SpotTrading`" is `ViewModelTrading.` An ExcelMVC session can have as many books as required, i.e. within the "Instance_Opened" event, you may need to have a switch statement on the view id and assign a desired model to each book view opened.

Implementing ExcelMVC View and Command Models

ExcelMVC creates an application's views in a tree structure:

- The root view of an ExcelMVC session is the singleton object of the App class, i.e. App.Instance.
- The App.Instance contains a list of Book views (App.Instance.Children). A book view is uniquely identified by its Id property, which is equal to the value of its "ExcelMVC" custom property or by its Name property, which is equal to its file name (e.g. SpotTrading.xlsx).
- A book view contains a list of Sheet views (Book.Children). A sheet view is identified by its Id or Name property, which is equal to its Excel tab name. ExcelMVC does not require a view model to be assigned to a book view.
- A sheet view contains a list of ExcelMVC Tables and/or Forms, identified by their Id or Name property, which are equal to the last dot name of their binding names. For example, if a table is named in the binding range as "ExcelMVC.Table.CcyPairs", then its corresponding view name is "CcyPairs". ExcelMVC does not require a view model to be assigned to a sheet view.

- A table view is bound to a collection of objects. The collection class must implement the System.Connections.IEnumerable interface and the System.Collections.Specialized. INotifyCollectionChanged interface if change notification is required after initial binding.
- A form view is bound to a single object, which needs to implement System.ComponentModel.INotifyPropertyChanged if change notification is required after initial binding.
- An ExcelMVC command raises mouse click events to the Clicked handler or to its Model object derived from System.Windows.Input.ICommand.

Completing the FX Spot Trading Sample

The ViewModelTrading class shown below is the root view model of the FX spot trading sample. The comments within the code describes how view models are assigned to views, and how command models are assigned to commands.

```
namespace SpotTrading.ApplicationModels
{
    using System.Linq;
    using BusinessModels;
    using CommandSinks;
    using ExcelMvc.Controls;
    using ExcelMvc.Views;
    using ViewModels;

    public class ViewModelTrading
    {
        public ViewModelTrading(View book)
        {
            // static ccy pair table (OneWayToSource)
            var tblCcyPair = (Table)book.Find("ExcelMvc.Table.CcyPairs");
            var pairs = new CcyPairs(tblCcyPair.MaxItemsToBind);
            tblCcyPair.Model = pairs;

            // static ccy list (OneWay)
            var tblCcys = book.Find("ExcelMvc.Table.Ccys");
            tblCcys.Model = pairs.Ccys;

            // exchange rates
            var tblRates = book.Find("ExcelMvc.Table.Rates");
            var rates = new ViewModelExchangeRates(new ExchangeRates(pairs));
            tblRates.Model = rates;

            // auto rate command
            var cmd = book.FindCommand("ExcelMvc.Command.AutoRate");
            cmd.Model = new CommandSinkAutoRate(rates);
            cmd.ClickedCaption = "Stop Simulation";

            // deal form
            var deal = new ViewModelDeal(rates);
            book.Find("ExcelMvc.Form.Deal").Model = deal;
            book.FindCommand("ExcelMvc.Command.InsideMode").Clicked += (x, y) =>
            {
                deal.IsInsideTrading = System.Convert.ToBoolean(((Command)x).Value);
            };

            // position table
            var tblPositions = (Table)book.Find("ExcelMvc.Table.Positions");
            var positions = new ViewModelPositions(tblPositions.MaxItemsToBind);
            tblPositions.Model = positions;
            book.FindCommand("ExcelMvc.Command.Reset").Clicked += (x, y) =>
positions.Reset();

            // manual deal command
            book.FindCommand("ExcelMvc.Command.ManualDeal").Model = new
CommandSinkManualDeal(deal, positions, rates);

            var dealing = new ViewModelDealing(pairs.Ccys.ToList(), deal, positions,
rates);
            cmd = book.FindCommand("ExcelMvc.Command.AutoDeal");
            cmd.Model = new CommandSinkAutoDeal(dealing);
            cmd.ClickedCaption = "Stop Auto-Deal";
        }
    }
}
```

If you are interested in understanding how other parts of the sample solution are implemented, you will need to dive deeper into the solution and go through the business model layer and the view model layer in details.

Otherwise, you are ready to run the FX spot trading program.

Launching an ExcelMVC Application

To run an ExcelMVC application, all is required is to start Excel, open ExcelMVC.Addin.xll or ExcelMVC.Addin (x64).dll (must reside in the application's folder) and then open the applications' workbooks.

For example, the following command file starts the FX spot trading sample from its debug build folder.

```
C:

CD  “ExcelMvc\Examples\SpotTrading\SpotTrading\bin\Debug”

REM for 64 bit Excel (switch “x” is to start a new Excel process)

Excel /x “ExcelMvc.Addin (x64).xll” “SpotTrading.xlsx”

REM for 32 bit Excel
```

Excel “ExcelMvc.Addin.xll” “SpotTrading.xlsx”

An ExcelMVC application typically has the following files in its application folder.

- ExcelMvc.dll
- ExcelMvc.Addin.xll or ExcelMvc.Addin (x64).xll
- Application specific assemblies
- Application configuration file (ExcelMVC pickups the first “*.dll.config” in the application’s folder as its configuration file)
- Microsoft.Office.Interop.Excel.dll, office.dll and Microsoft.Vbe.Interop.dll, which are only required if your application targets .NET 3.5.

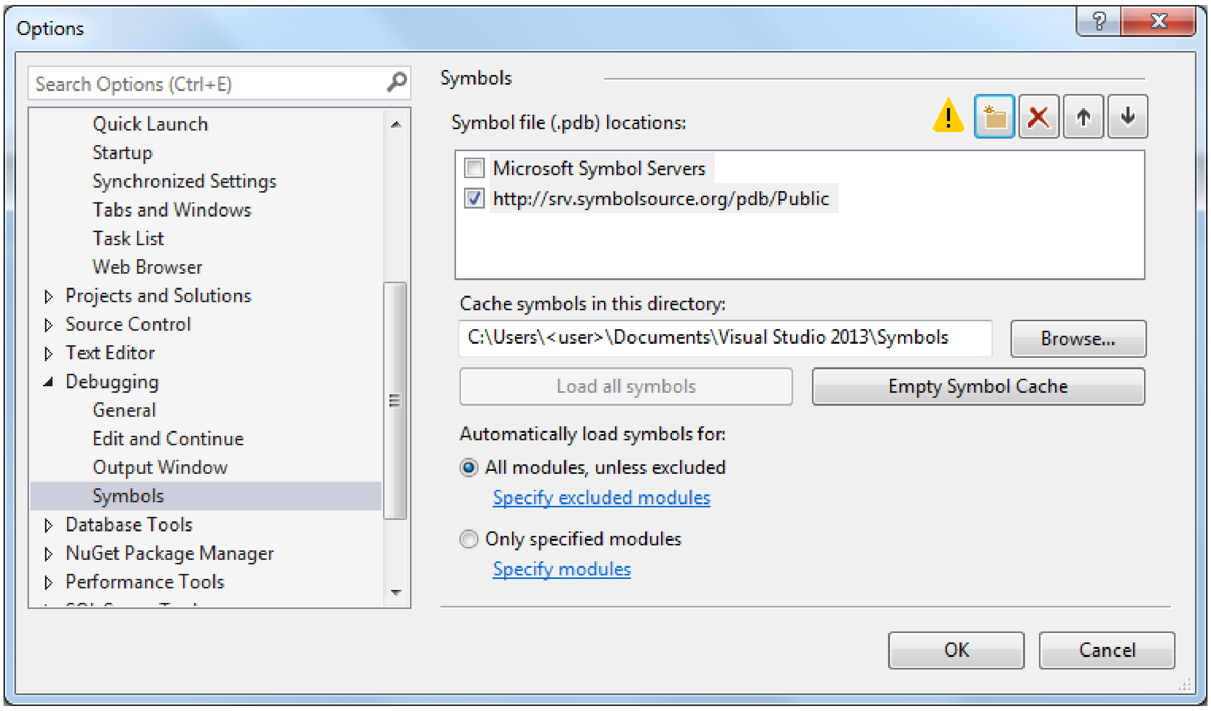
If you have added ExcelMVC to your solution using the ExcelMVC NuGet package, and you have set the file property “Copy to Output Directory” to “Always” for ExcelMvc.Addin.xll, ExcelMvc.Addin (x64).xll and the application’s workbooks, then the build out folder will have everything you need to run the application.

Debugging an ExcelMVC Application

If you have installed the ExcelMVC NuGet package to your .NET solution (see section *Setting up an ExcelMVC .NET solution* above), Visual Studio 2013 can download the source code and symbol files (*.pdb) on demand for you. That means, with a little bit of configuration to Visual Studio, you can step through ExcelMVC during debugging. The configuration needs to be done only once.

The sources of ExcelMVC have been uploaded to symbolsource.org, they will be updated with every new release of ExcelMVC. To configure Visual Studio to download sources and symbols from this server, follow these instructions:

- In Visual Studio, go to Tools | Options | Debugging | General.
- Uncheck “Enable Just My Code”.
- Uncheck “Require source files to exactly match the original version”
- Go to Tools | Options | Debugging | Symbols.
- Add the symbol server `http://srv.symbolsource.org/pdb/Public`
- Select a folder for the symbol cache. Have a look at the following picture to check your symbol server settings.


Options

Search Options (Ctrl+E)

Quick Launch
Startup
Synchronized Settings
Tabs and Windows
Task List
Web Browser
Projects and Solutions
Source Control
Text Editor
Debugging
General
Edit and Continue
Output Window
Symbols
Database Tools
NuGet Package Manager
Performance Tools

Symbols

Symbol file (.pdb) locations:

[ ] Microsoft Symbol Servers
[✓] http://srv.symbolsource.org/pdb/Public

Cache symbols in this directory:
C:\Users\<user>\Documents\Visual Studio 2013\Symbols

Browse...

Load all symbols

Empty Symbol Cache

Automatically load symbols for:
[● All modules, unless excluded]
Specify excluded modules
○ Only specified modules
Specify modules

OK Cancel

Now, if you run the spot trading sample in Excel as described above, go to Visual Studio debugger and click on Debug | Attach to Process ..., then select EXCEL and start stepping through ExcelMVC code.

Starting ExcelMVC Development Now!

If you have been developing applications in WPF, WinForm or Silverlight, we hope by now you realise you can develop these applications in Excel much easier and faster with ExcelMVC.

Stop trying to replicate Excel UI in your applications. We know it is extremely hard to design and implement good UI.

And start using ExcelMVC now so that you can focus on your applications' business layers.

The ExcelMVC Team

Peter Gu, Sydney

Wolfgang Stamm, Germany