# Getting Started with ExcelMVC

Like what Silverlight or WPF (Windows Presentation Foundation) does, ExcelMVC facilitates a clear separation between your application’s business objects (Models), its user interfaces (Views) and its controllers (View Models). Welcome to the world of developing Excel applications using the MVVM ([more](http://en.wikipedia.org/wiki/Model_View_ViewModel)) pattern!

With ExcelMVC, you define your views in Excel (as opposed to in Xmal with Silverlight or WPF) and bind their view models (implemented in any .NET languages) to them. Your views are Excel specific, but your view models and your business objects 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.

Can you imagine how difficult it would be to replicate some of the Excel UI functions, e.g. formatting, charting, printing, import /export etc., with WPF or Silverlight?

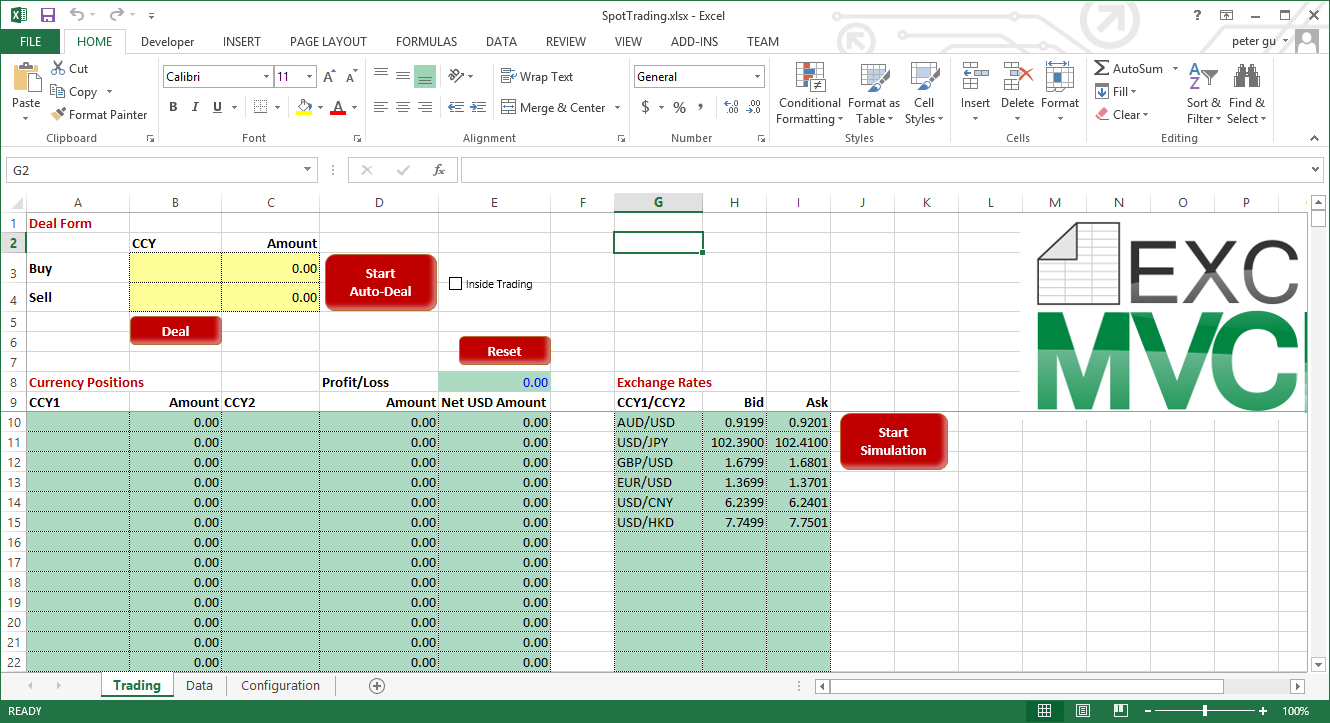
Can you also imagine with ExcelMVC, it becomes possible and easier to test your Excel applications? ExcelMVC removes the need of having complicated formulas and macros in your Excel workbooks.

So without further ado, let’s get started.

# Downloading the 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](http://sourceforge.net/projects/excelmvc/files/latest/download/ExcelMVC.zip) 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 a file.
3. Double click on the “Run.cmd” file to start the sample application. You will see an Excel screen showing below.
4. Click on “Start Auto-Deal” and “Start Simulation”, and you are on your way of making (or losing) loads money from the foreign exchange spot market.
5. Navigate to other tabs on the sheet to get a basic idea how ExcelMVC glues things together. Press Alt-F11 to see there is absolute no VB code behind



# Creating an ExcelMVC application

A typical ExcelMVC application is made up by two parts: the Application’s UI (one or more workbooks) and its code behind (one or more .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 glued together by ExcelMVC.

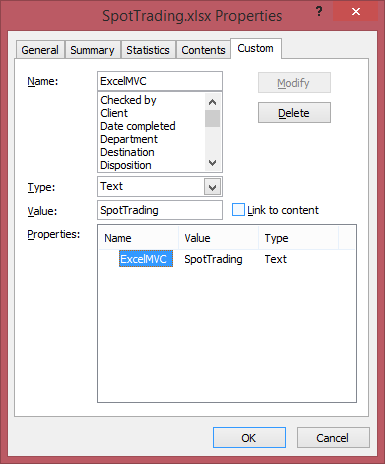
We will use the FX spot trading sample to work through the process of creating a ExcelMVC project.

## Designing ExcelMVC workbooks

#### Identifying ExcelMVC workbooks

An ExcelMVC workbook is identified by a custom property “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).



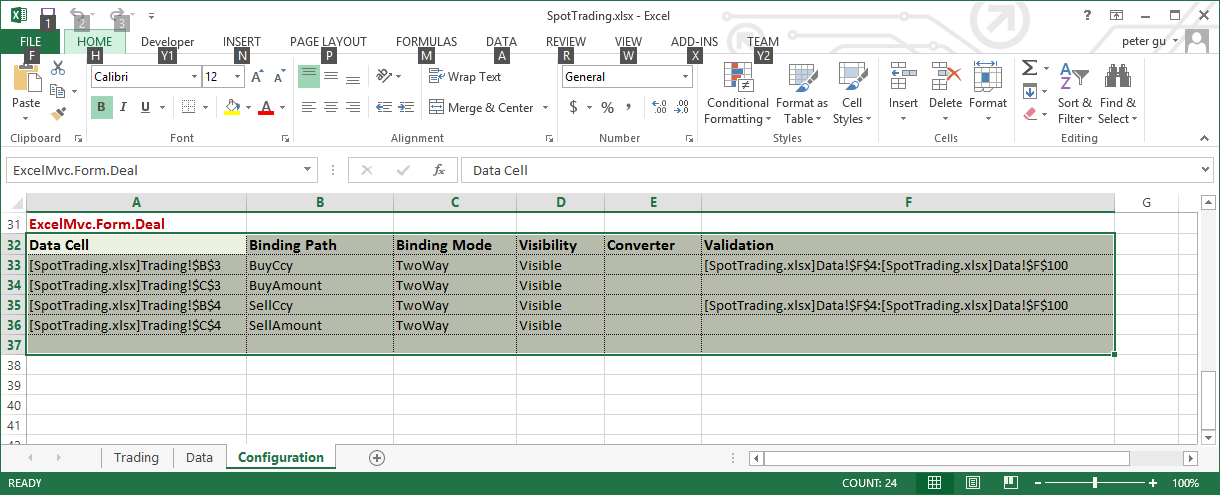
Later in this tutorial, you will see how ExcelMVC notifies your application whenever a workbook is about to be opened in your Excel session. You use this property to tell if a workbook belongs to your ExcelMVC application.

#### Defining ExcelMVC Forms and Tables

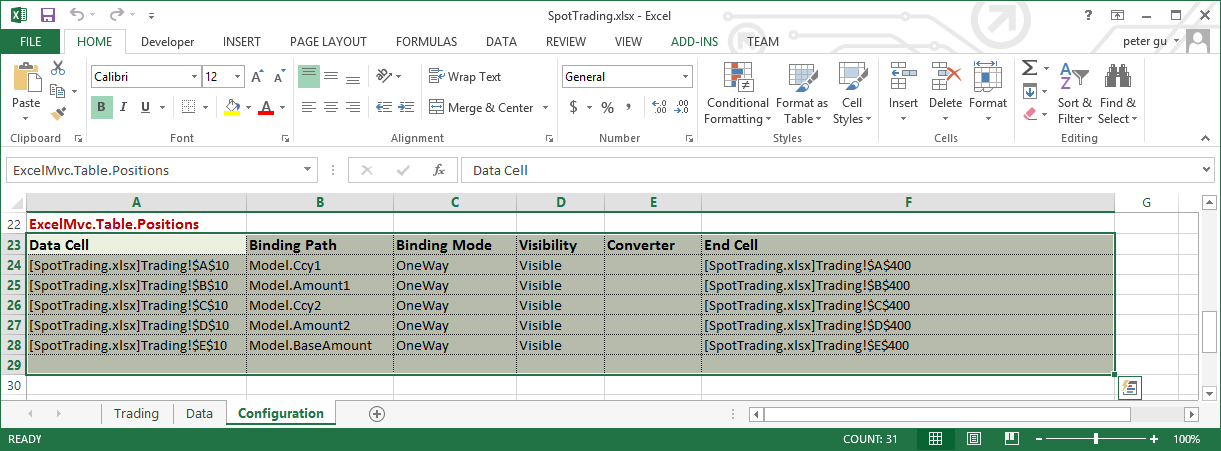
ExcelMVC currently supports two types of views, Form and Table. A form facilitates the binding between the properties of an object and Excel cells, while a table facilitates the binding between a collection of objects and Excel rows or columns.

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.



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.



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 lists headings required by ExcelMVC.

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

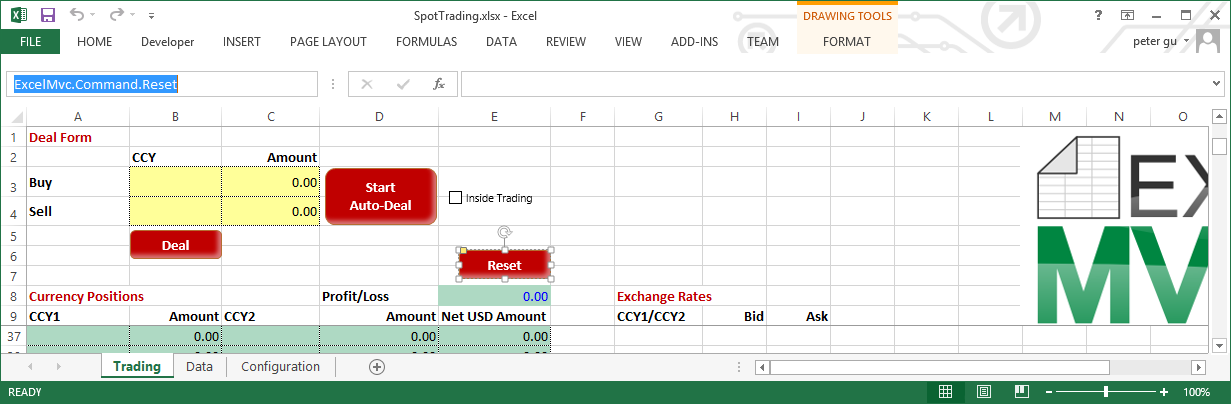
A landscape table binds each object in its view model to an Excel column. To define a landscape table, simply specify the Data Path of each binding on a single Excel column.

#### 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 does not support ActiveX controls, which are provided for use with VBA. As one of the objectives of ExcelMVC is to reduce the amount of VBA code behind Excel workbooks, therefore it is probably not a real limitation that ActiveX controls are not supported by ExcelMVC.

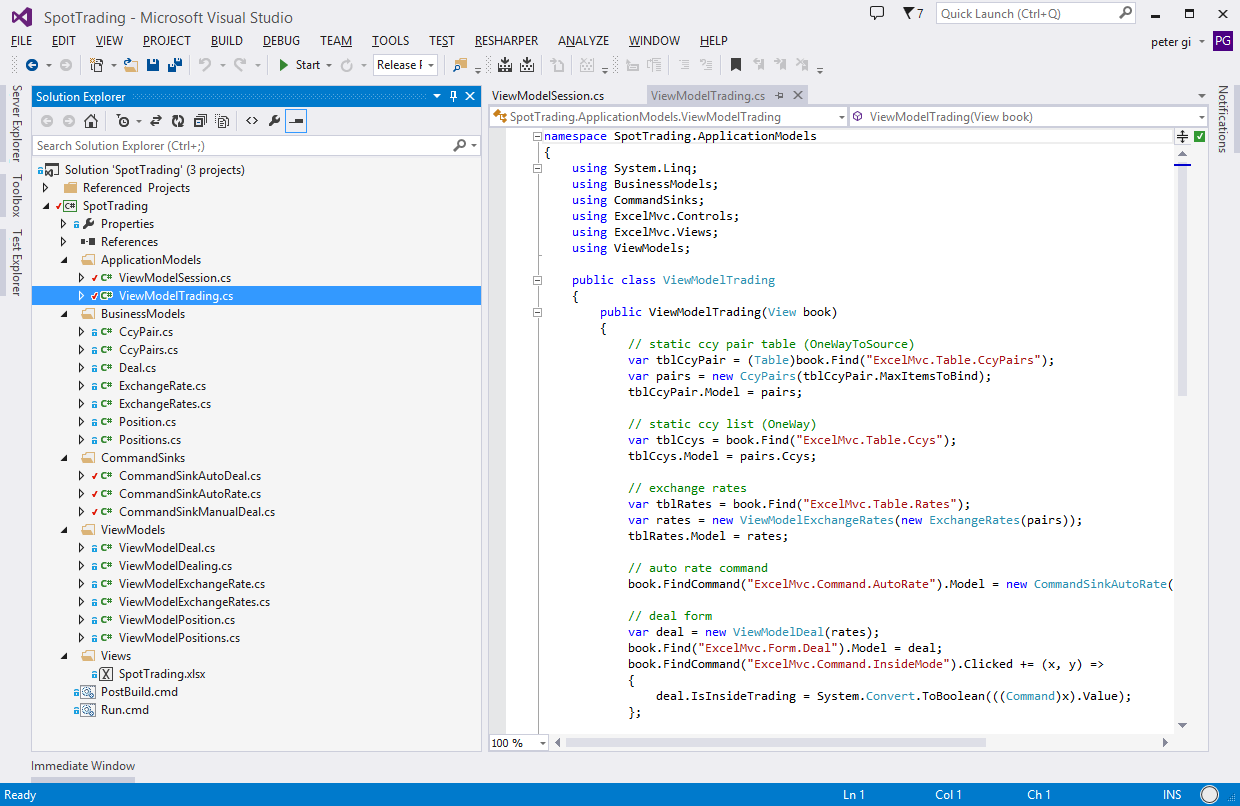
## Setting up an ExcelMVC .NET Solution

An ExcelMVC applications typically requires four types of projects:

1. Business model assemblies, with each containing classes 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 standard Window forms (WPF or WinForm) required. 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”) to 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 classes listed under 1 to step 3.)

1. Create application’s business object 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 the ExcelMVC reference 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 seatings under Tools | NuGet Package Manager | Package Manager Settings). Or, you can reference the ExcelMVC.csproj located under the “source\ExcelMvc” folder. The screenshot below shows the FX spot trading sample solution (you can find it under the “samples\trading” folder).



1. 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 use ExcelMVC views only, then you don’t need this assembly.
2. 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 differences between setting up an ordinary .NET solution and setting up an ExcelMVC solution.

## Creating an ExcelMVC Session

ExcelMVC binds an application’s views to their models through its session object derived from ExcelMVC.Runtime.ISession. An application’s session object is created by ExcelMVC automatically when the application is stared with ExcelMVC.Addin.xll located in the application’s directory is loaded into Excel.

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 used in the code section “SpotTrading” must be equal to the value of custom property “ExcelMVC”, 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)

{

// cancel out for non-ExcelMvc books

if (args.View.Id.CompareOrdinalIgnoreCase(BookId) != 0)

args.Cancel();

}

void Instance\_Opened(object sender, ViewEventArgs args)

{

// create book model

if (args.View.Id.CompareOrdinalIgnoreCase(BookId) == 0)

args.View.Model = new ViewModelTrading(args.View);

}

void Instance\_Closing(object sender, ViewEventArgs args)

{

// cancel close

// args.Cancel();

}

void Instance\_Closed(object sender, ViewEventArgs args)

{

// remove view models

args.View.Model = null;

}

public void Dispose()

{

}

}

}

In the FX spot trading sample, for the “SpotTrading”, 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 Book views (App.Instance.Children). A book view is uniquely identified 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 Sheet views (Book.Children). A sheet view is identified by its Id or Name property, each 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, each of them is identified by their Id or Name property, which is equal to the last dot name of its binding name. 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 connection class must implement the System.Connections.IEnumerable interface and the System.Collections.Specialized. INotifyCollectionChanged interface if change notification is required after the initial binding.
* A form view is bound to a single object, which needs to implement System.ComponentModel.INotifyPropertyChanged if change notification is required after the initial binding.
* An ExcelMVC command can sink mouse click events to the Clicked handler or to a Model object derived from System.Windows.Input.ICommand, or to both.

## 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 their views, and command models/sinks are assigned their 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 project are implemented, you will need to dive deeper into the solution and go through the business models layer and the view models layer in details. Note you may implement some or all of classes differently. Otherwise, you are ready to try the FX spot trading sample.

## Launching an ExcelMVC Application

To be continued.

You are now ready to start your own ExcelMVC development.