



# Adopt a Services Architecture

## Overview

Line-of business applications have incredible diversity. From scenarios to technologies, no two applications are exactly alike. Given the long lifetimes of typical line-of-business applications, it's always valuable to plan ahead and architect components to be as future-proof as possible. Unfortunately, there is also pressure to deliver as soon as possible, and sometimes development teams don't know what their users will need a year or two down the road. As a result, it's often best to architect an application in such a way that components are abstracted as services. There has been a lot of coverage about service-oriented architecture for multi-tiered applications, but in this lab we'll focus on thinking about client applications and the benefits of abstracting internal application components with a similar service mindset. These services could be as simple as using interface-based programming with dependency injection so that each component relies on a service instead of a concrete implementation. At another level, Model-View-ViewModel (MVVM) could be thought of as an abstracted service where a layer of objects provides access to data and functionality that could be implemented in virtually any way.

In our scenario, we have an existing expense reporting client that has been developed as a WCF application. While an effort has been made to follow best practices, you can tell from a review of the codebase that it has gone through several rounds of revision, and parts of the application have been updated while others have not. Some components use solid practices, while others still rely on their previous implementation. However, this is usually how it goes in the real world. Fortunately, there's some great stuff we can do to improve the long-term quality and maintainability of the application. We'll even add a new feature while we're at it.

## Objectives

In this hands-on lab, you will learn how to:

- Set up the expenses infrastructure
- Create some unit tests for our client application
- Add a viewmodel feature and expose it in a view

## Prerequisites

The following is required to complete this hands-on lab:

- Microsoft Visual Studio 2013

- SQL Express 2012 with Tools (SQL Server Management Studio)
- Microsoft Web Platform Installer 2.6 or later

## Setup

We'll set up the environment in Exercise 1.

## Exercises

This hands-on lab includes the following exercises:

1. Setting up the environment
2. Create a unit test project
3. Adding a new feature

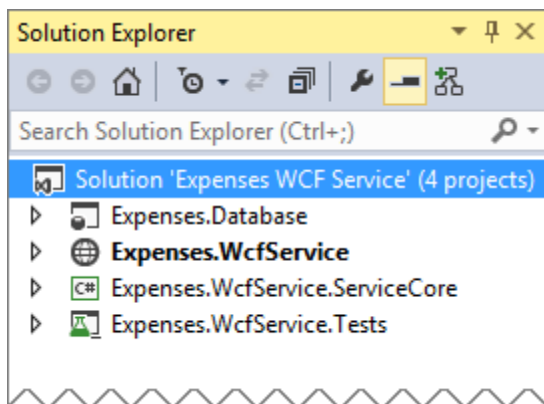
### Exercise 1: Setting up the environment

In this exercise, we'll go through the process of setting up our environment.

#### Task 1: Setting up the local database

In this task, we'll create the local database used by our expense reporting system.

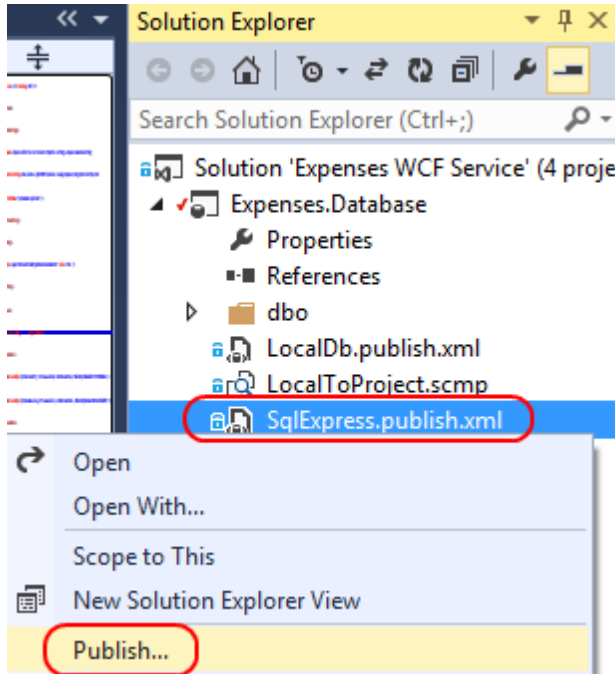
1. If needed, unzip the lab files to a convenient place on your system.
2. In Visual Studio 2013, open the **"Source\Begin\Expenses WCF Service\Expenses WCF Service.sln"** solution file.



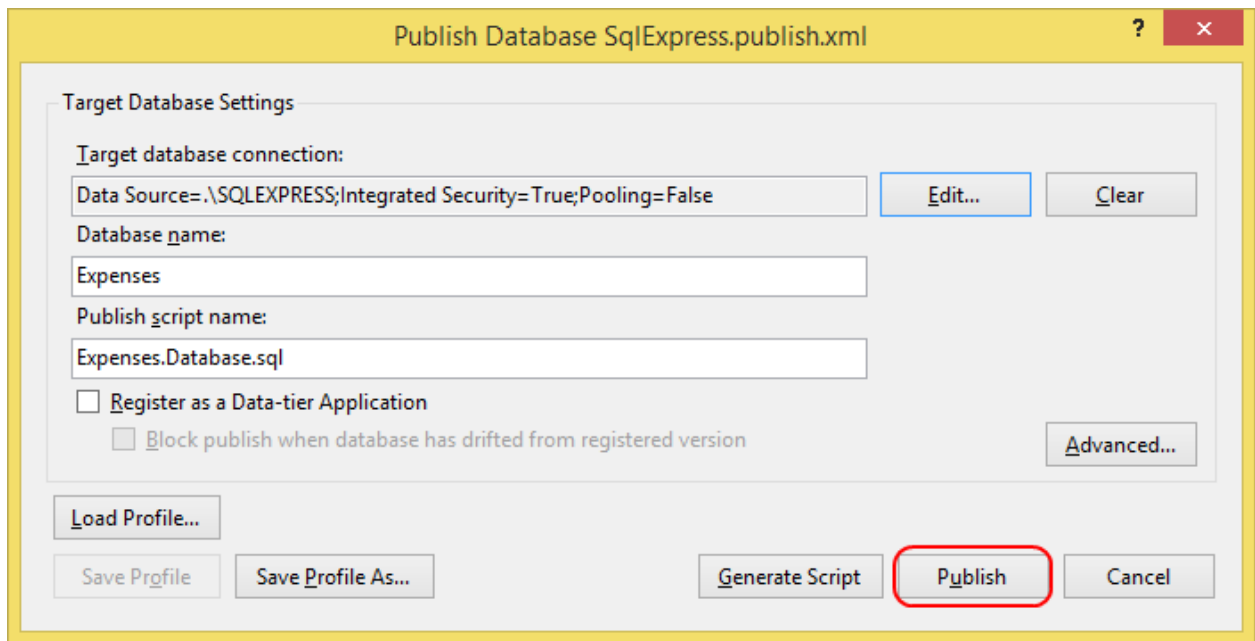
3. This solution contains:
  - a. **"Expenses.Database"** is a SQL Database project that defines the schema for the tables in the "Expenses" database. In a minute, we'll use this to publish the database to our SQL Server Express instance.
  - b. **"Expenses.WcfService"** contains the ASP.NET host for the **ExpenseService** implemented in the **"Expenses.WcfService.ServiceCore"** project
  - c. **"Expenses.WcfService.ServiceCore"** the services data and service contracts and the core service implementation. The service implementation uses a LINQ-TO-SQL DataContext to access the data in the SQL Database, but it could just as easily use Entity Framework, or some other storage layer to access expense data.

d. **"Expenses.WcfService.Tests"** is a collection of Unit Tests for the WCF service.

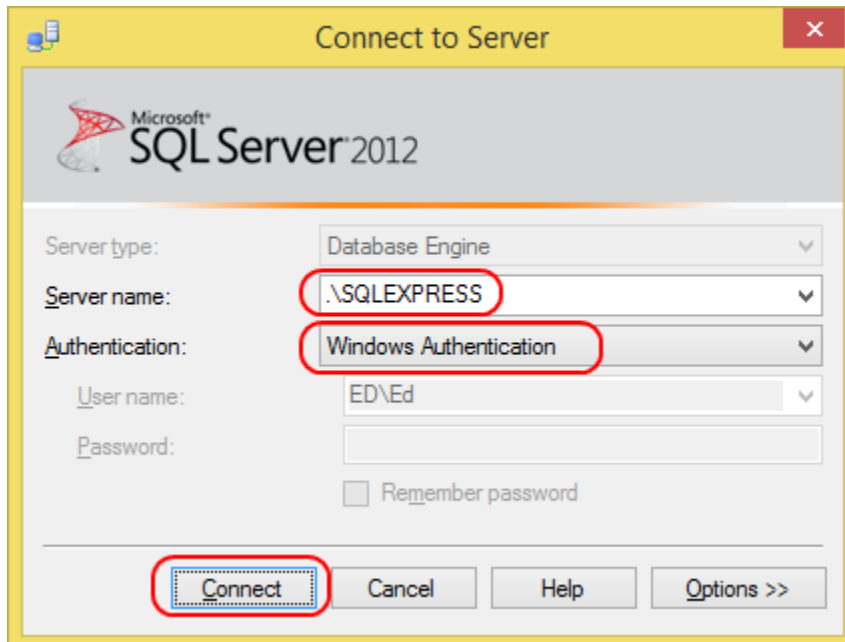
4. In the "Solution Explorer" window, expand the **"Expenses.Database"** project **right-click** the **SqlExpress.publish.xml** file in the **Expenses.Database** project and select **Publish...**.



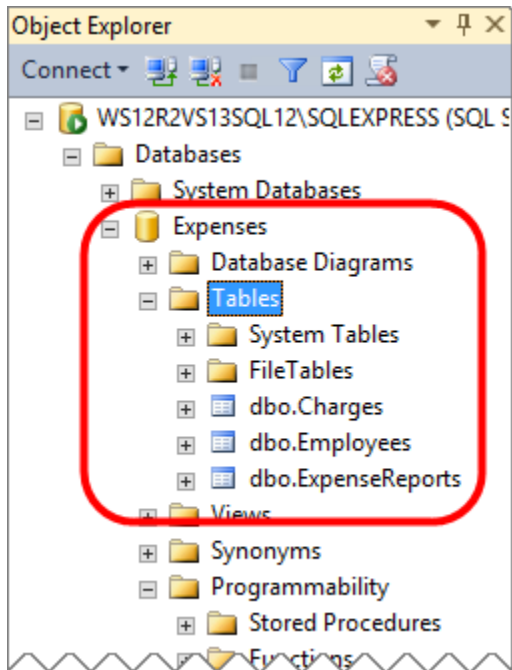
5. Assuming you have a SQL Express Server Express 2012 instance installed with the default name of **SQLEXPRESS** and that you have administrative access to that instance, the default values in the **"Publish Database SqlExpress.publish.xml"** widow should work. If needed you can change the connection settings to point to a more appropriate SQL Server instance. Be aware that if you do so, you will need to change the connection string in the web service's web.config file to point to the alternate location.
6. Click **Publish** to publish the database to your SQL Express instance. It should complete the deployment within seconds.



7. Launch **SQL Server Management Studio**. We'll use this tool to add the IIS user for our Web site to the SQL logins so it can access the database using Windows Authentication.
8. Connect to the local database using the **Server name** ".\SQLEXPRESS". Use **Windows Authentication** and click **Connect**.



9. Expand the **Databases | Expenses | Tables** node under the local server and verify that the **dbo.Chargers**, **dbo.Employees** and **dbo.ExpenseReports** tables are there.

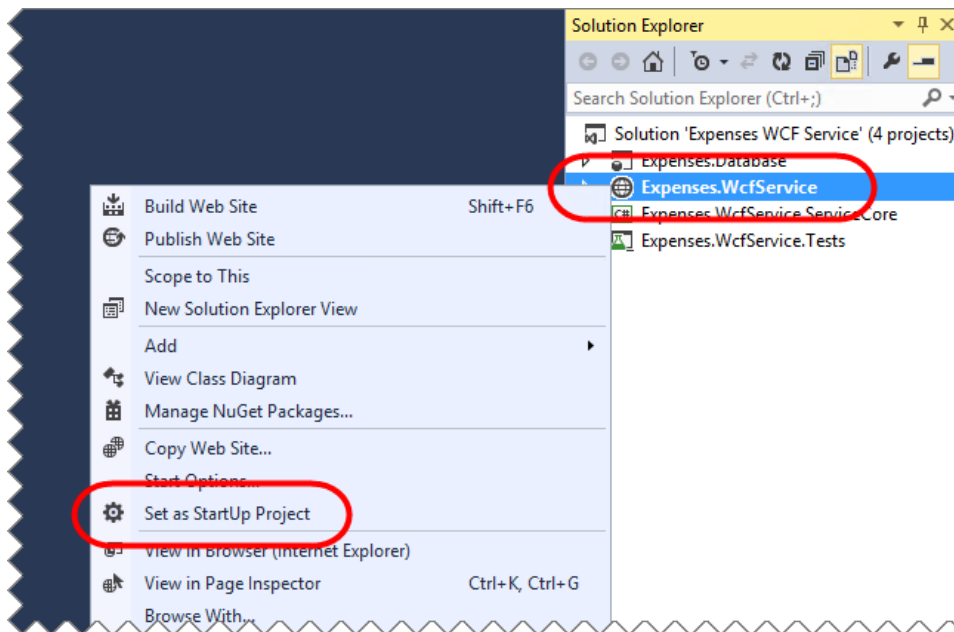


10. Close **SQL Server Management Studio**.

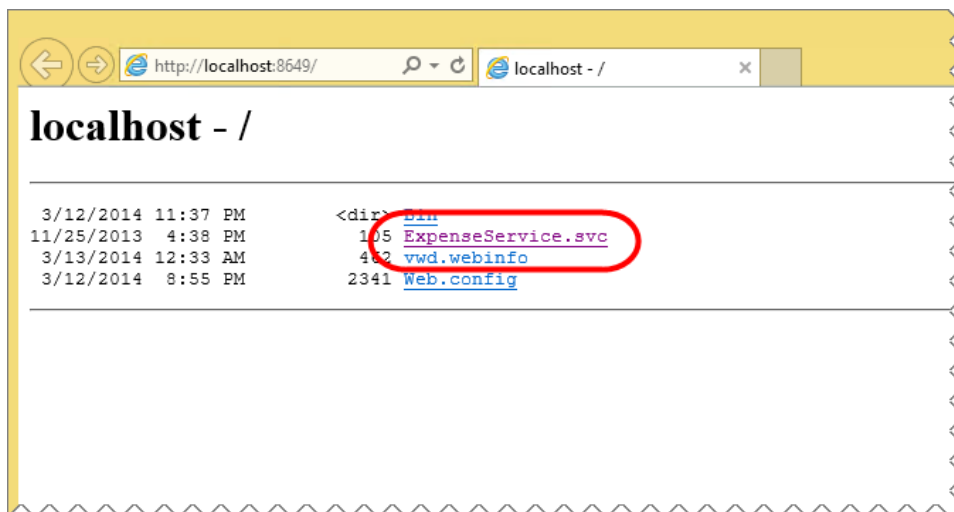
#### Task 2: Setting up the WCF service

In this task, we'll set up the WCF service that includes our server-side logic and data access. The service will be hosted by IIS Express by running it from within Visual Studio on your development workstation.

1. Back in Visual Studio, in the **"Solution Explorer"** window, ensure that the **"Expenses.WcfService"** project is the startup project by **right-clicking** on the project and selecting **"Set as Startup Project"** from the pop-up menu.



2. From the Visual Studio menu bar, select “**DEBUG**” | “**Start Debugging**”, or press the **F5** key to start a debug session.
3. The **Expenses.WcfService** website should be open in the browser. Click the link for the **ExpenseService.svc** web service



4. And verify that the “**ExpenseService Service**” test page loads successfully, then copy the URL for the service to the



5. **IMPORTANT!** Leave the browser window open so that the service is ready to receive requests, and leave the “Expenses WCF Service” solution open in Visual Studio

### Task 3: Touring the expenses WPF application

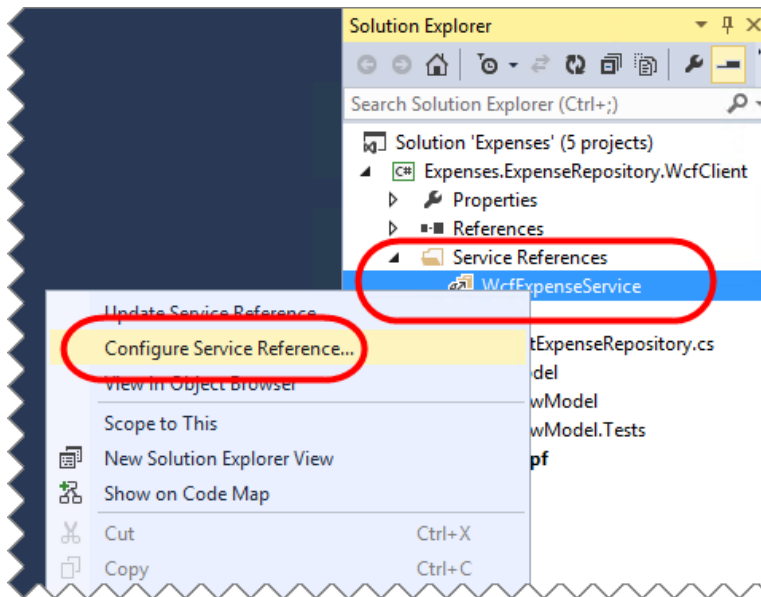
In this task, we’ll take a look at the features of our WPF application.

1. Open a SECOND COPY of Visual Studio 2013, and Open “\Source\Begin\Expenses WPF\Expenses.sln” solution from the lab files. This solution contains our WPF application, as well as its dependencies.
2. There are five projects in this solution:
  - a. **Expenses.Model** defines the data model classes, constants, and interfaces that are shared across projects. It is a portable class library that can be used in WPF, Windows Phone, or Windows Store applications.
  - b. **Expenses.ViewModel** implements most of the business logic as ViewModels and also includes some business-level utilities. It references the model project. It is a portable class library that can be used in WPF, Windows Phone, or Windows Store applications.
  - c. **Expenses.ExpenseRepository.WcfClient** is a project that implements an **IExpenseRepository** as defined in the model project. It contains all the logic for communicating with the WCF service. It is a portable class library that can be used in WPF, Windows Phone, or Windows Store applications.
  - d. **Expenses.Wpf** is our WPF application, and references the model, ViewModel, and WCF client projects.
  - e. **Expenses.ViewModel.Tests** is our testing project, which contains unit tests that target the ViewModel library.
3. In the **Expenses.Model** project, open **IExpenseRepository.cs**. This interface defines the various methods needed to work with the data in our system. Note that the implementation could take

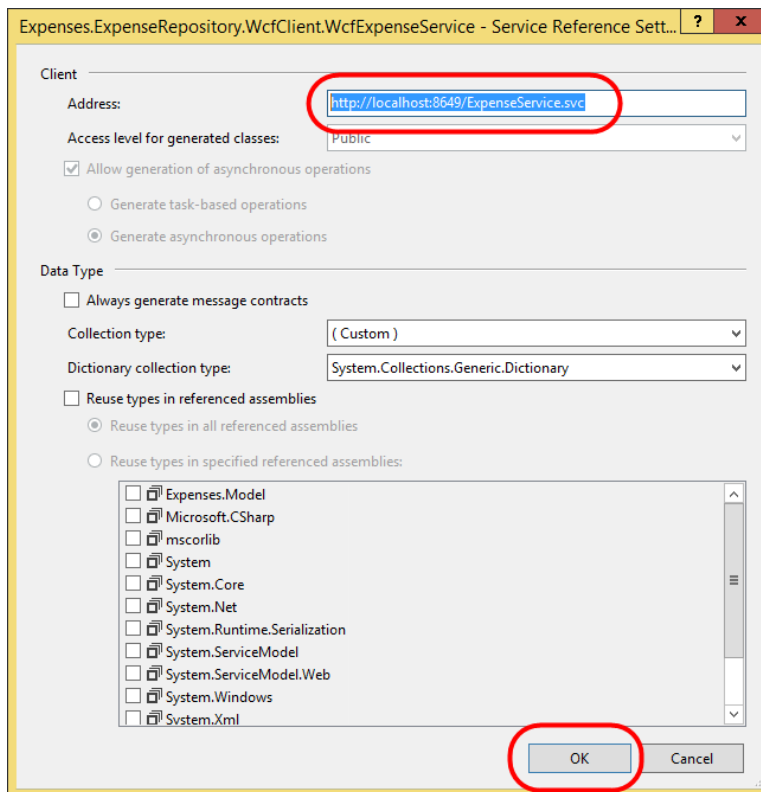
virtually any form, such as a SQL store, a WCF service, a file-based system, etc. And just as important, by abstracting away the implementation, we can build software at higher layers that take a dependency on the interface, which we can mock up for testing purposes. Another important detail to note is that the methods here all return Task objects. This allows the caller to use the **await** keyword in order to enjoy the benefits of simpler asynchronous programming.

4. In the **Expenses.ExpenseRepository.WcfClient** project, open **WcfClientExpenseRepository.cs**. This is an implementation of our **IExpenseRepository** that knows how to fulfill the methods using our backend WCF service. You'll note there are two constructors. The overloaded takes a required parameter for the URL of the service. The default constructor is marked private so that it is not available for creating new objects without the configuration parameter.
5. In the **Model VMs** folders of the **Expenses.ViewModel** project, open the **ChargeViewModel.cs** file. This is a ViewModel that derives from our **ViewModelBase** class that includes some base services, as well as implements **INotifyDataErrorInfo** for data validation and error binding in XAML. If you scroll down to the constructor, you can see that we're loading the **IExpenseRepository** from a global service locator. The service locator pattern is controversial, but is how this particular dependency is loaded. Depending on how the service locator was set, this could be our **WcfClientExpenseRepository** from earlier, a mocked-up **IExpenseRepository** from our test layer, or a completely different implementation we haven't yet considered.
6. In the **Expenses.Wpf** project, open **App.xaml.cs**. In **OnStartup** you can see where we load the WCF service URL from the app settings, as well as create the **WcfClientExpenseRepository** and set it in the **ServiceLocator**. If we wanted to change the **IExpenseRepository** used for the entire application, we could do so here.
7. We need to make sure that the **Expenses.ExpenseRepository.WcfClient** project is pointing to our version of the WCF Service that we just tested. In the **Solution Explorer** window, expand the **"Expenses.ExpenseRepository.WcfClient" | "Service References"** node. Then, right-click on the **"WcfExpenseService"** service reference, then select **"Configure Service Reference..."** from the pop-up menu.

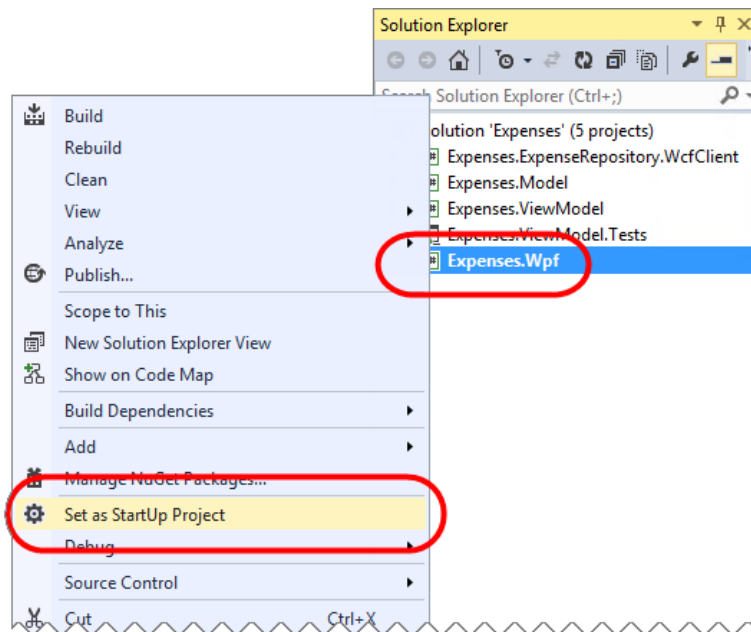




8. Then, paste the URL for the service that you copied from the browser previously into the “**Address:**” field, and click “**OK**” (Make sure that the WCF Service is still running in the other copy of Visual Studio and that the browser window we opened for the service previously is still open. Keep it open throughout this lab).



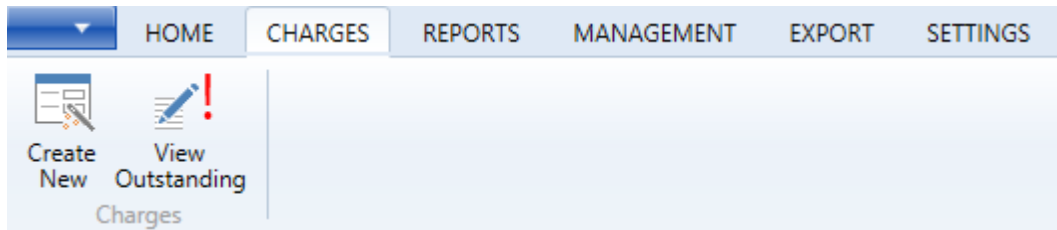
9. Ensure that the **Expenses.Wpf** project is the startup project by right-clicking on it in the Solution Explorer window, and selecting “**Set as StartUp Project**” from the pop-up menu:



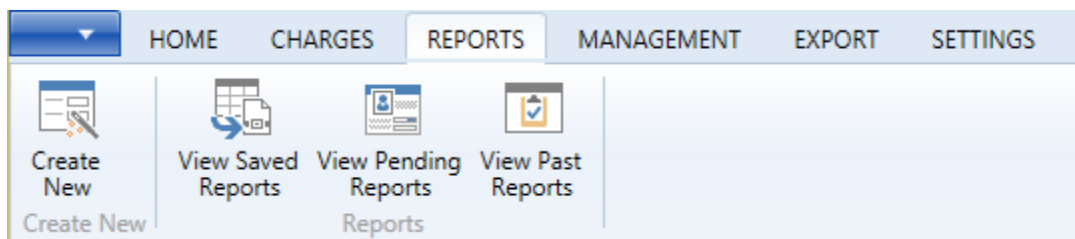
10. Press **F5** to build and launch the application. It will automatically log you in as the user “Robert Green” with the alias “rogreen”. The first time you log in, it will generate some sample data for your account. The default view is the “Summary” that highlights some general details about your current status.



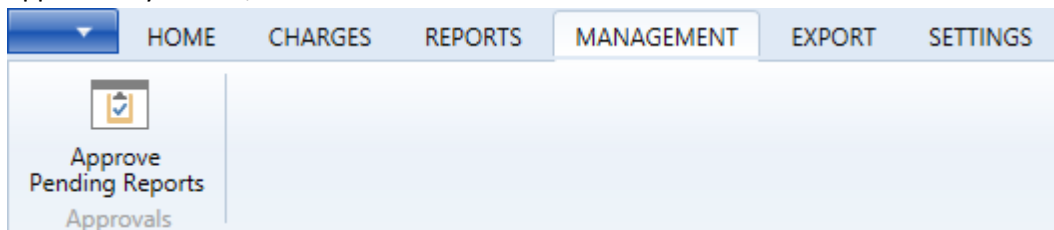
11. There are three basic features to this application: managing your charges, managing your expense reports, and approving subordinate expense reports. Under **Charges**, you have the ability to manually create new charges and the ability to view outstanding charges that have not been assigned to an expense report.



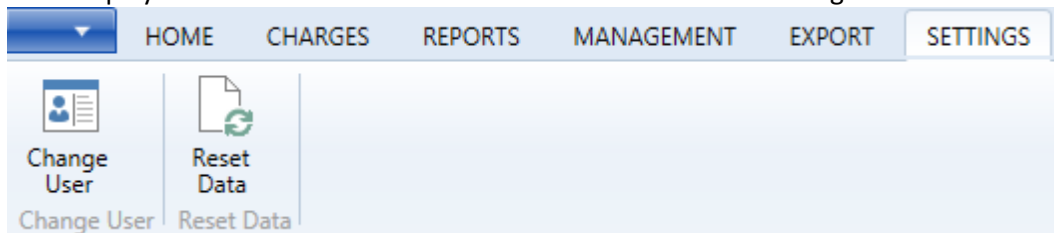
12. On the **Reports** tab, you can create a new report, as well as view reports that you've saved but not yet submitted, reports you've submitted but have not been resolved, and reports that have been resolved.



13. On the **Management** tab you can find reports that your subordinates have submitted for your approval. By default, there are two.



14. The **Export** tab functionality has not been implemented for this lab. The **Settings** tab provides two options. **Change User** isn't used in our labs, so you can ignore it. **Reset Data**, however, will reset the database to its original state and repopulate pending reports, etc. This is a very useful option if you want to play around with the features and then revert back to the original data at the end.



15. Feel free to explore the app. When done, be sure to click **Reset Data** before moving to the next exercise.

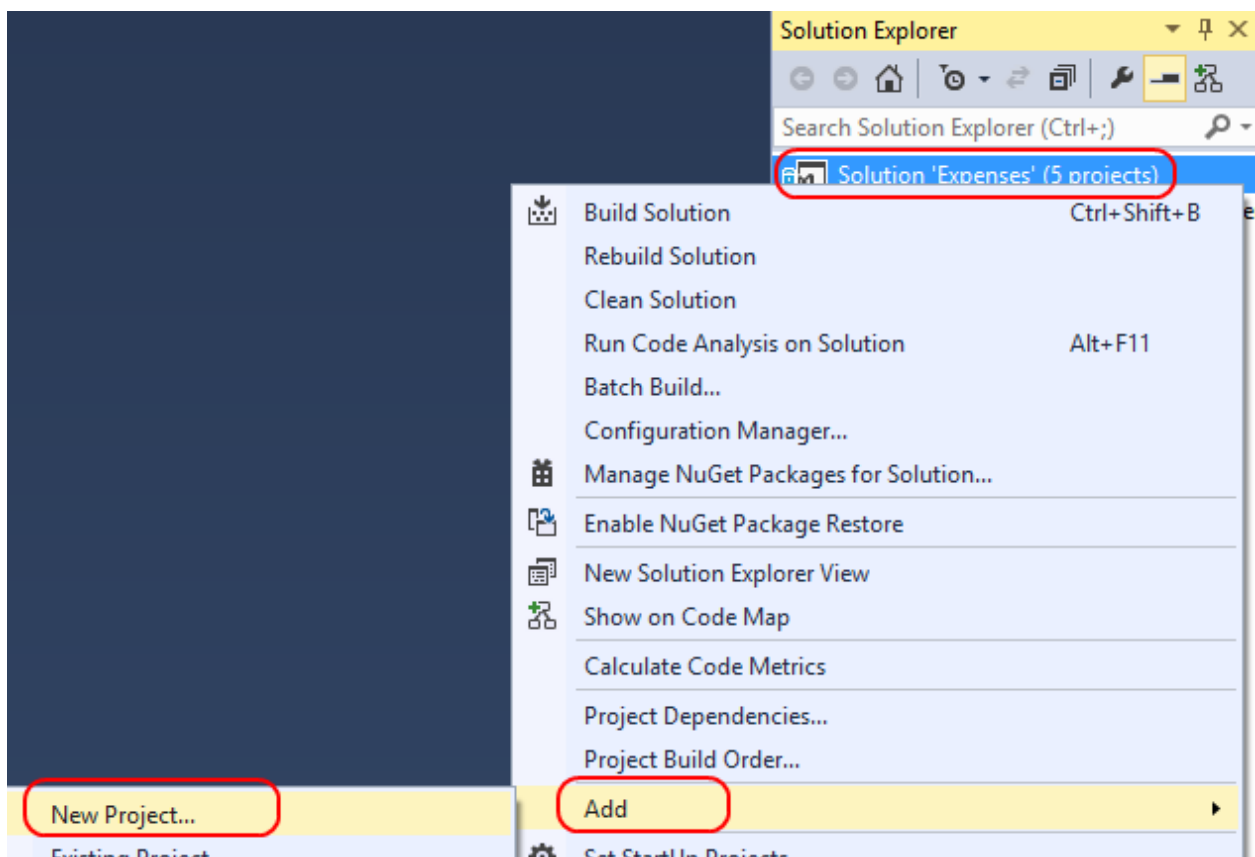
## Exercise 2: Adding a unit test project

In this exercise, we'll go through the process of adding a unit test project for our ViewModel project. One of the great benefits of dependency injection and inversion-of-control is that they encourage software that has a more testable surface area. This in turn aids the development process by making progress more apparent and avoiding regression issues.

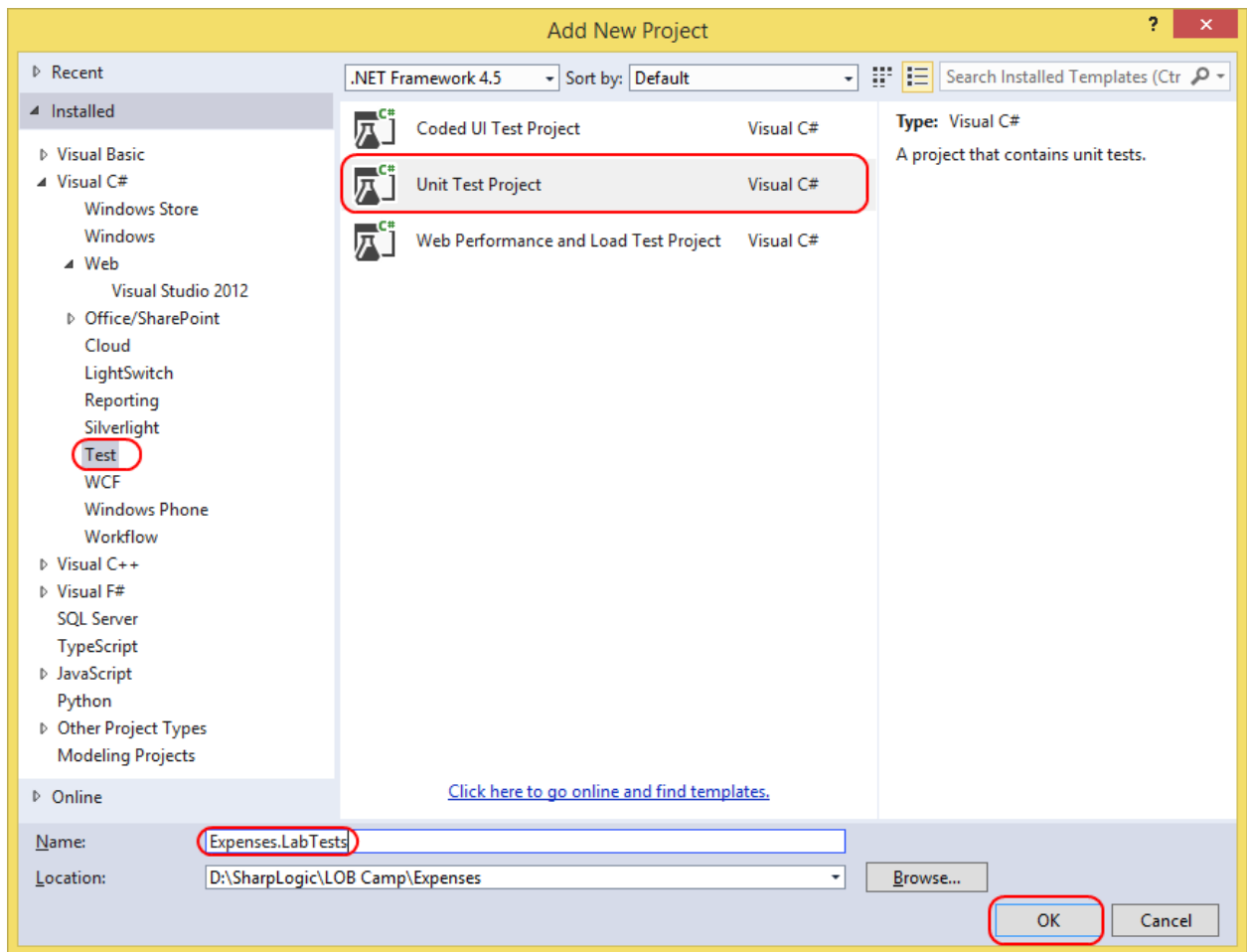
### Task 1: Setting up the project

In this task, we'll set up the unit test project. This will include the project creation, as well as adding the necessary references. We'll also add some **Fakes**, which are like mocks, to help us with dependency injection.

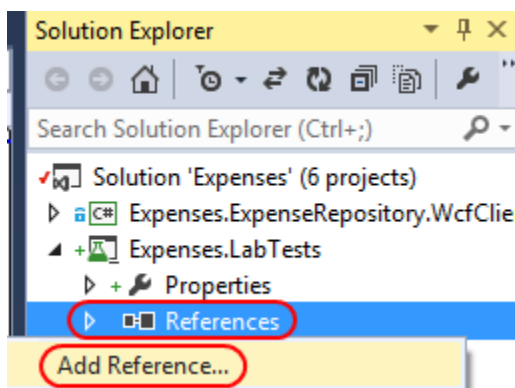
1. In the copy of Visual Studio that has the "**Expenses**" solution with the WPF Client application we just tested in it, right-click the **Solution** node and select **Add | New Project....**



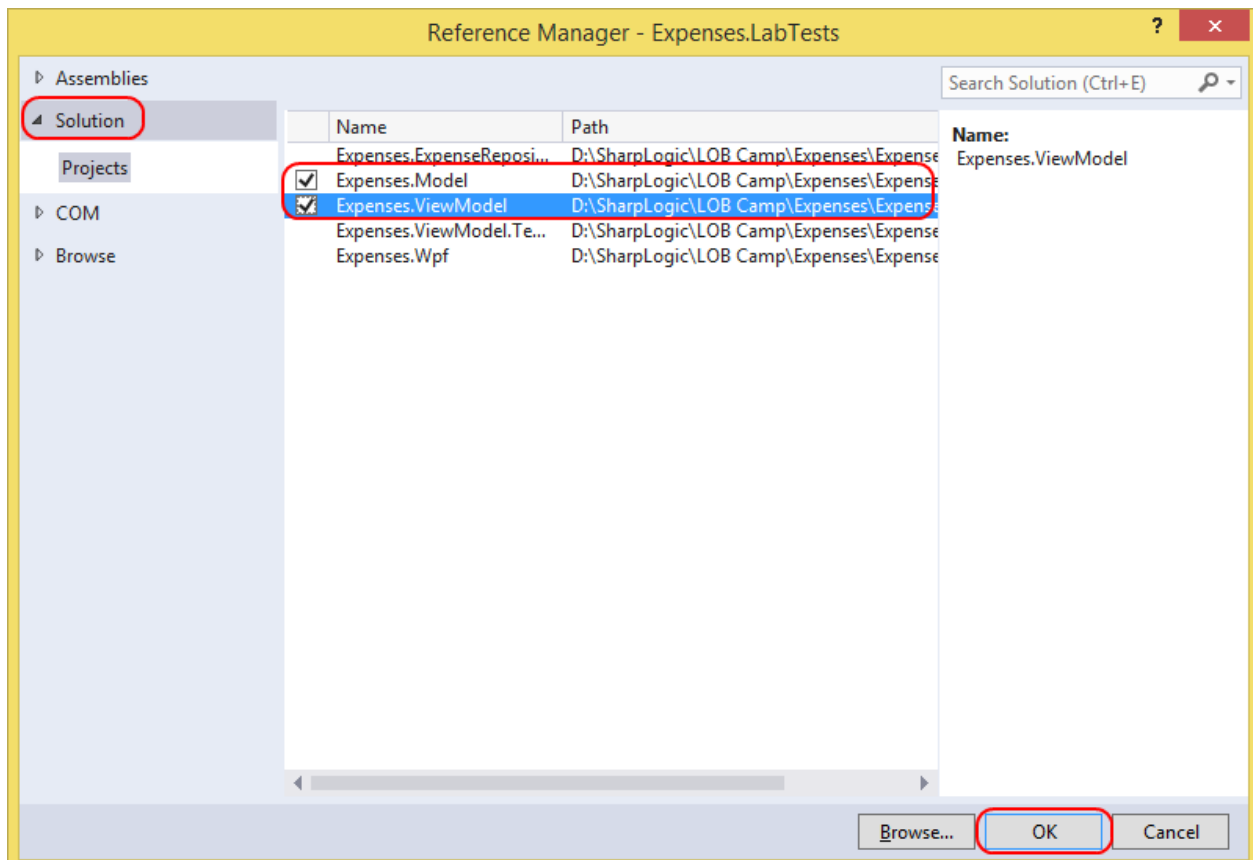
2. Select **Test** from under the **Visual C#** node in the left project types panel. Select the **Unit Test Project** template and set the **Name** to **Expenses.LabTests**. Click **OK** to create the project.



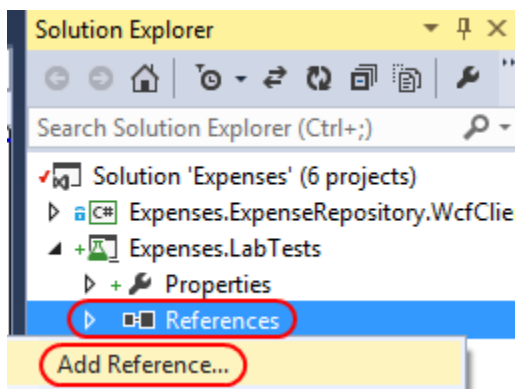
3. The project will set up the basic things we need to create a unit test. However, we'll also need to add some references to the solution projects we want to include as part of our testing. Right-click the **References** node under the new **Expenses.LabTests** project in the **Solution Explorer** and select **Add Reference...**



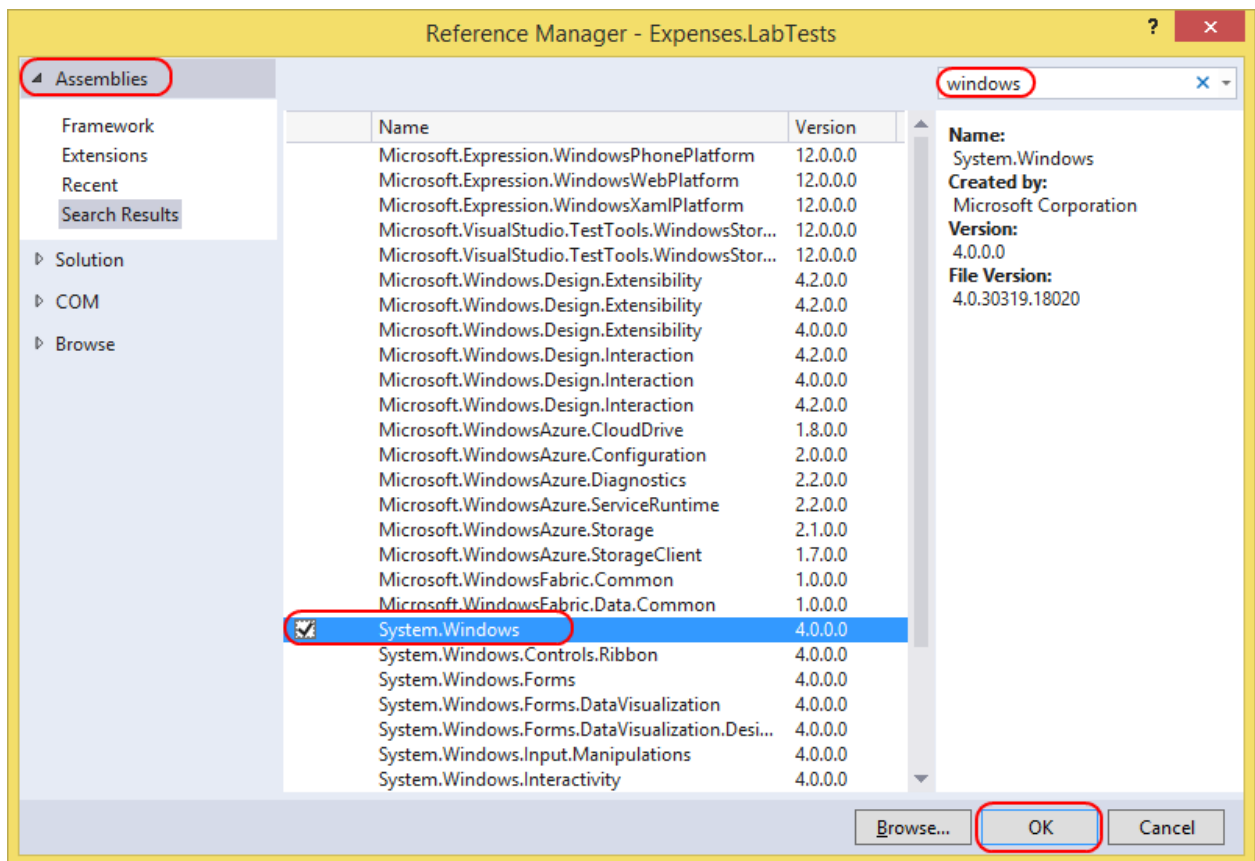
4. Select the **Solution** list from the left panel and check **Expenses.Model** and **Expenses.ViewModel** in the assemblies list. Click **OK** to add references to them.



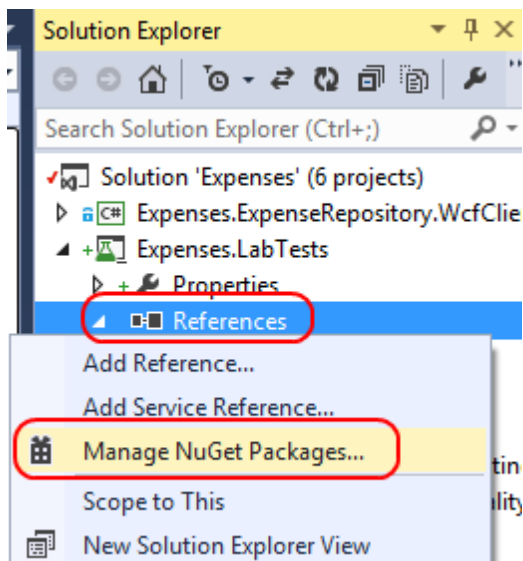
- Next, we'll add a reference to **System.Windows** as needed by the ViewModel project. Right-click the **References** node under the new **Expenses.LabTests** project in the **Solution Explorer** and select **Add Reference....**



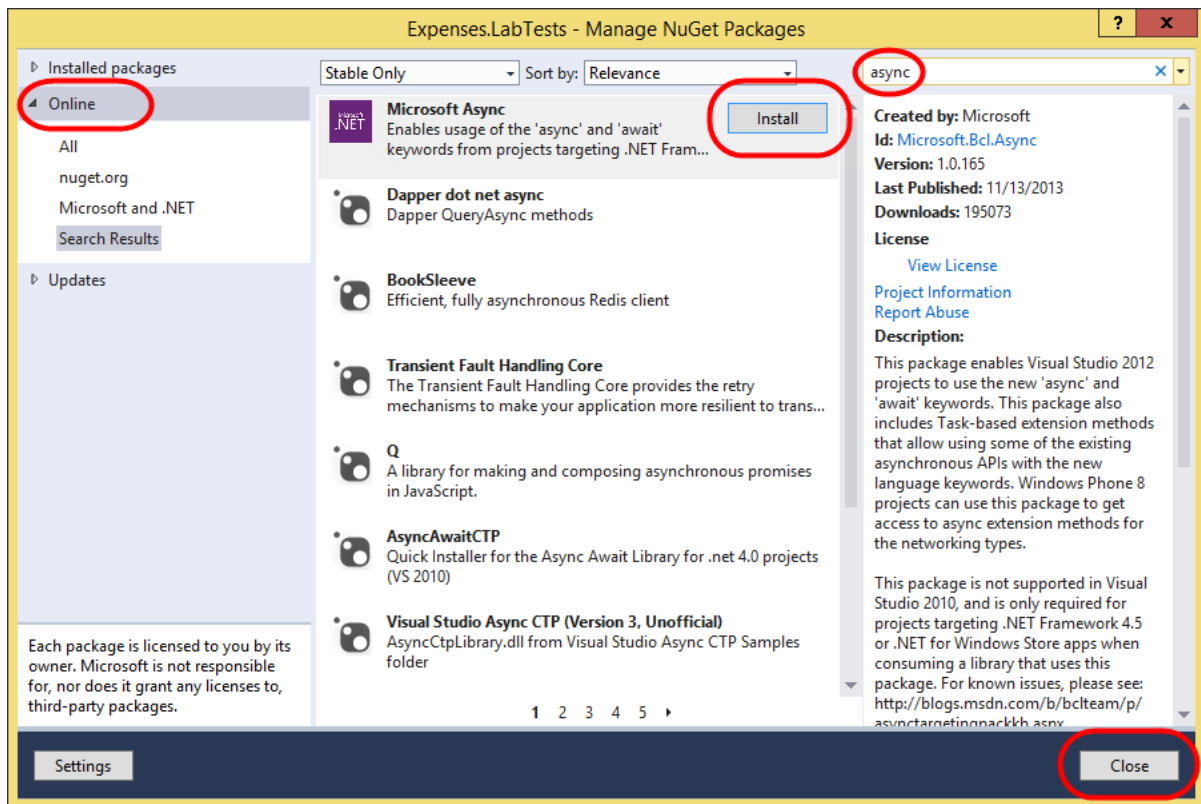
- Select the **Assemblies** list from the left panel and type "**windows**" in the search box in the top right corner. Check **System.Windows** in the assemblies list and click **OK** to add the reference.



7. We'll also need to add **NuGet** packages to support the **Task** model used by these libraries. Right-click the **References** node under the test project and select **Manage NuGet Packages....**

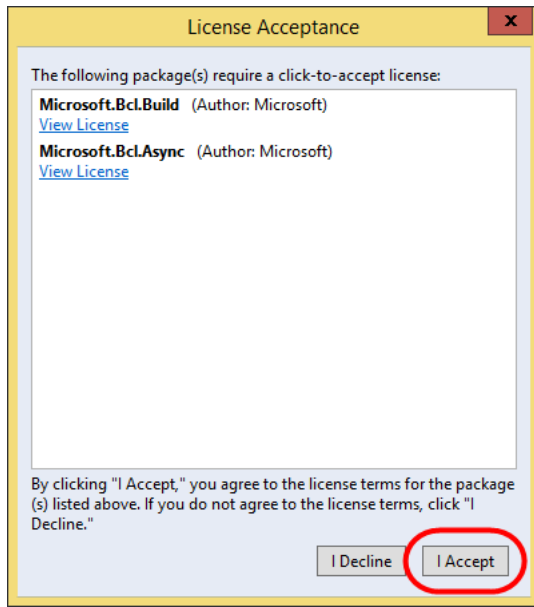


8. Select the **Online** option from the left pane. Next, type “**async**” into the search box in the top right corner. When the search results load, click **Install** next to the **Async for .NET Framework 4, Silverlight 4...** package created by **Microsoft**.



9. In the “**License Acceptance**” window click “**Accept**”, then once the packages have installed, click “**Close**” in the “**Expenses.LabTests – Manage NuGet Packages**” window.

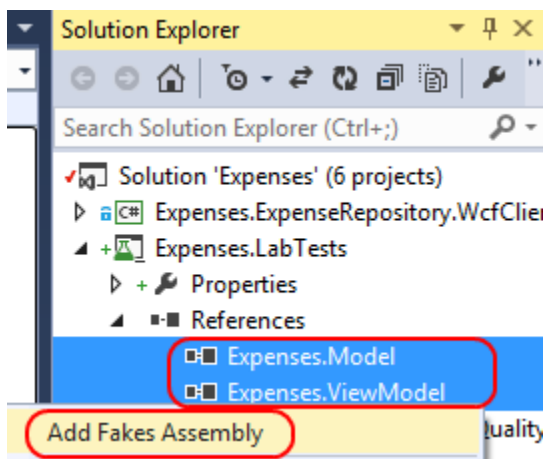




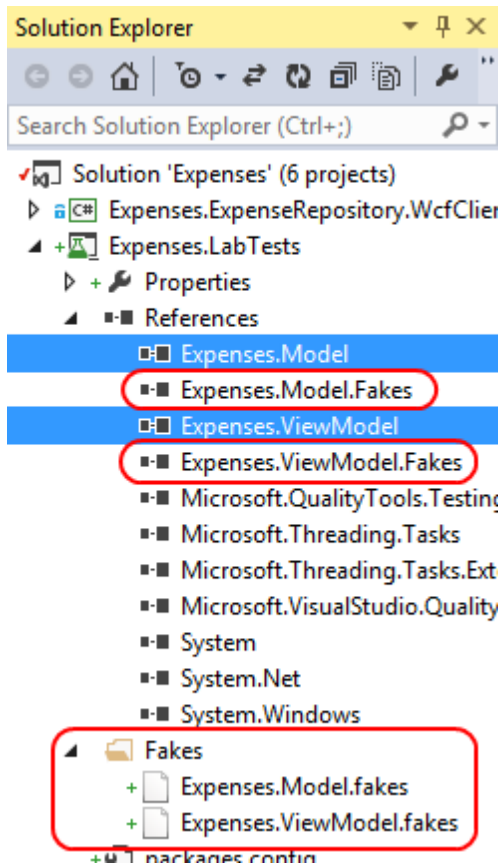
## Task 2: Creating Fakes assemblies

In this task, we'll add some Fakes assemblies to make it easier to work with referenced classes and interfaces.

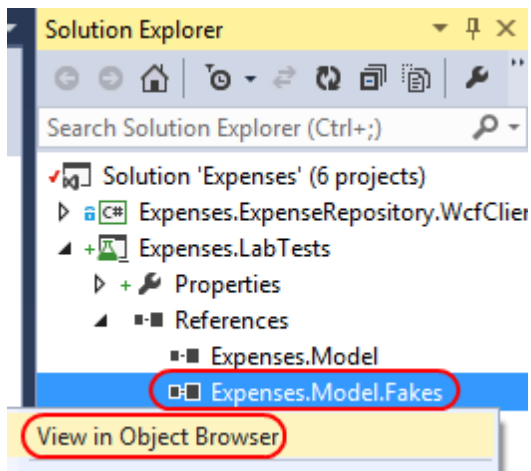
1. We want to write a test for our **ChargeViewModel**, which relies on two interfaces that it picks up from the **ServiceLocator: IViewService** and **IExpenseRepository**. Rather than build out complete implementations of these for testing purposes, we can rely on **Microsoft Fakes** to support basic stubbing.
2. Under the **References** node of the test project, select both **Expenses.Model** and **Expenses.ViewModel** (hold the **Shift** key to multi-select). Right-click the selection and select **Add Fakes Assembly**.



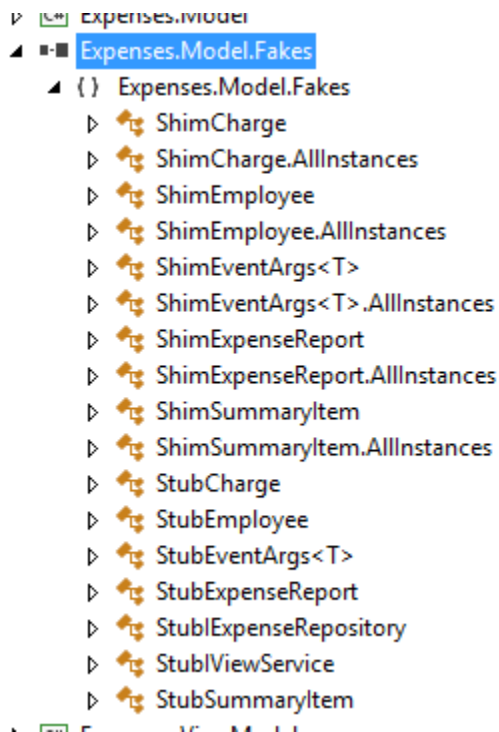
3. Visual Studio will now generate "fake" versions of the types in the assembly so that you can easily override their behavior and/or implement their required methods and properties.



4. Right-click the **Expenses.Model.Fakes** assembly and select **View in Object Browser**.



5. If you expand the assembly and namespace in the object browser, you can see that it's full of shim and stub objects. The “**Shim**” objects give you the ability to finely control the behavior of classes defined in the original library. The “**Stub**” classes are implementations of the interfaces of the original library that you can easily insert your own methods as needed.



### Task 3: Writing and running a test

In this task, we'll write and run our test.

1. Now let's turn our attention to building a unit test. Open **UnitTest1.cs** and find **TestMethod1**. To start off with, let's add the using declarations at the top so we can resolve the types we need.

```
using Expenses.Model;  
using Expenses.Model.Fakes;  
using Expenses.ViewModel;  
using Expenses.ViewModel.Fakes;  
using System.Threading.Tasks;
```

2. Since we're looking to perform some basic testing for the **ChargeViewModel**, let's paste in the initial code. The comments explain the code. Paste the following code in as the body of the **TestMethod1** method.

```
// Create a new ChargeViewModel.  
ChargeViewModel chargeViewModel = new ChargeViewModel();  
  
// Make sure it defaults to a ChargeId of 0.  
Assert.AreEqual(0, chargeViewModel.ChargeId);  
  
// Load the charge with the ChargeId of 1.  
await chargeViewModel.LoadAsync(1);
```

```
// Confirm the ViewModel's ChargeId is 1.
Assert.AreEqual(1, chargeViewModel.ChargeId);
```

3. The first thing you'll notice from the code above is that it raises an error on the **await** statement. This is because **await** requires the method to be marked as **async**, and it should almost always return some sort of **Task**. Update the method definition to the line below.

```
async public Task TestMethod1()
```

4. Now the code will build. It's also important to note that the Visual Studio testing engine supports async testing just like it does synchronous testing, which makes life much easier for developers. However, just because the code will build doesn't mean it's ready to test. We provide three dependencies to this ViewModel via our **ServiceLocator**, so we'll need to set those up before we can test. First, we'll add an **INavigationService**. Since this isn't actually used in our code path, we can just give it the default stub object. Add this code to the **beginning** of the **TestMethod1** method just above the code we pasted in previously.

```
ServiceLocator.Current.SetService<INavigationService>(
    new StubINavigationService());
```

5. Next, we need to provide the **ServiceLocator** with an **IViewService**. Our code path will use the **ExecuteBusyActionAsync** method on this interface, so we'll need to provide a method for the **ExecuteBusyActionAsyncFuncOfTask** stub. It won't do anything special, so this method simply runs the function it's provided. You don't need to understand this code because it's not relevant as far as our testing is concerned. Add the code below directly after the code from the previous step.

```
StubIViewService viewService =
    new StubIViewService()
    {
        ExecuteBusyActionAsyncFuncOfTask =
            async (Func<Task> func) =>
            {
                await func();
            }
    };
ServiceLocator.Current.SetService<IViewService>(viewService);
```

6. However, one service that does play an important role for our test is the **IExpenseRepository**. Since this is the object that is used to load the specified charge, we need to provide a method to run when its **GetChargeAsync** gets called. Fortunately, there is a **GetChargeAsyncInt32** stub that we can fill out. Paste this code after the code from the previous step. At this point, the next line of code after this block should be the line that creates the **ChargeViewModel** that we added in first.

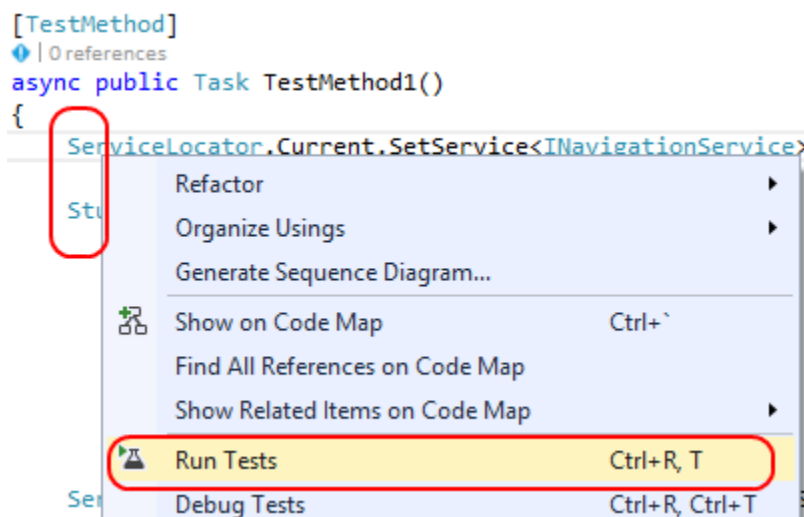
```
StubIExpenseRepository repository =
    new StubIExpenseRepository()
```

```

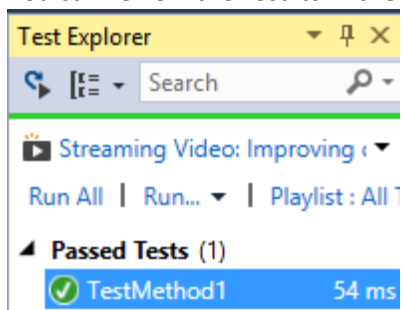
{
    GetChargeAsyncInt32 =
        (chargeId) =>
        {
            return Task.FromResult(
                new Charge()
                {
                    ChargeId = chargeId,
                });
        }
};
ServiceLocator.Current.SetService<IExpenseRepository>(repository);

```

7. Open the **Test Explorer** by selecting **Test | Windows | Test Explorer** from the main menu.
8. Right-click inside the body of the test method and select **Run Tests**. The run should complete almost immediately.



9. You can review the results in the **Test Explorer**.



### Exercise 3: Adding a new feature

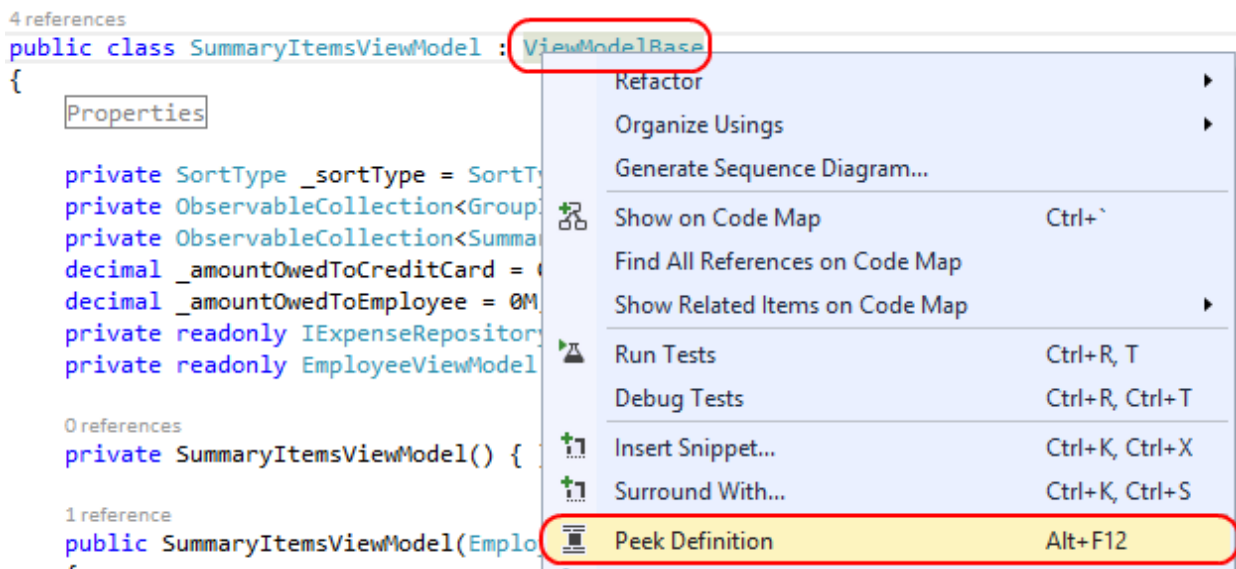
In this exercise, we'll go through the process of adding a new feature to our app. We'll add a color-coding feature in the `ViewModel` portable class library and databind it to the background of a grid in the

WPF app. If we were working purely in a WPF app, this would be a simple thing to accomplish. However, we're using a PCL to contain as much business logic as possible so we can share it across different device targets. As a result, we're going to need to apply a few techniques to make it all run smoothly. Note that this scenario is a little contrived for the purposes of this demo and there may be more efficient ways to accomplish our goal.

#### Task 1: Adding the ViewModel property

In this task, we'll add the color coding property to the **SummaryItemsViewModel**. This will allow the user to quickly see whether they have outstanding reports to approve (red) or if they're all caught up (green). However, since we're working with a PCL, there is no support for the **Brush** or **Color** objects, so we'll need to work around this using a **Tuple<byte, byte, byte>** to return the RGB data.

1. Do the following steps in the copy of Visual Studio that has the "Expenses" solution with our WPF client in it.
2. In the **Expenses.ViewModel** project, open the **SummaryItemsViewModel.cs** class.
3. At the class definition, right-click the inheritance reference to **ViewModelBase** and select **Peek Definition**. This will bring up a peek view of **ViewModelBase**.



4. **ViewModelBase** implements **INotifyPropertyChanged** and manages how we notify any databinding clients of changes in property value. Since we're working in a PCL, we don't have **DependencyObject** to derive from, so we need to do a little extra work. Fortunately, all this functionality carries with our assembly—in binary form—across Windows Phone and Windows Store apps as well. Another neat trick employed here is the use of **Expressions** to perform the property change notifications. Since the notifications themselves use **strings**, there is the risk of error whenever property names are changed or modified, such as by refactoring. Thanks to the **NotifyOfPropertyChange<TProperty>** method, child classes can pass a reference to the property itself, from which the property name is retrieved at runtime. Press **Esc** to close the **Peek Definition** window.

5. Add the following code directly inside the class definition so that it's the first property in the class. This is the property that represents the background color we want to display in the summary view where we're summarizing the status of the reports waiting for the user to approve. The **Tuple** is a nifty object that allows us to group a set of values together, which is perfect for our purposes in this lab.

```
public Tuple<byte, byte, byte> MyApprovalsBackgroundColor
{
    get { return _myApprovalsBackgroundColor; }
    set
    {
        this._myApprovalsBackgroundColor = value;
        this.NotifyOfPropertyChanged(() =>
            this.MyApprovalsBackgroundColor);
    }
}
private Tuple<byte, byte, byte> _myApprovalsBackgroundColor;
```

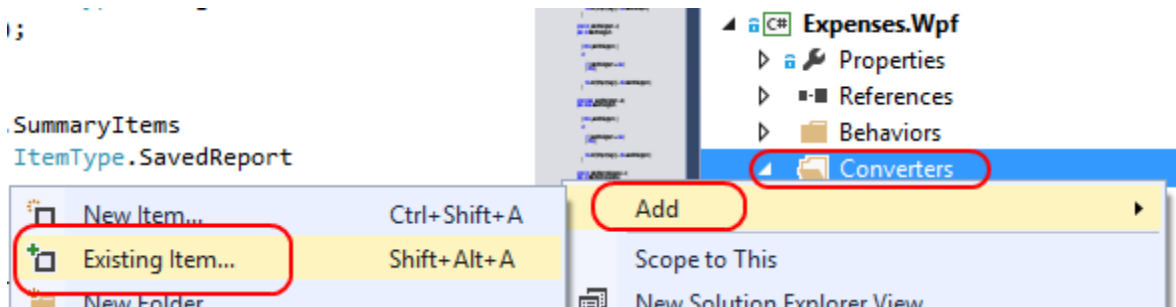
6. Find the **GetSummaryItems** method. Scroll down to the bottom and add the following code directly before the "return this.SummaryItems" line. This code sets our preferred background color to a light red if there are any pending reports waiting for the user's approval, or light green if they are all caught up.

```
if (this.NumberOfReportsNeedingApproval > 0)
{
    this.MyApprovalsBackgroundColor =
        new Tuple<byte, byte, byte>(255, 127, 127);
}
else
{
    this.MyApprovalsBackgroundColor =
        new Tuple<byte, byte, byte>(0, 255, 127);
}
```

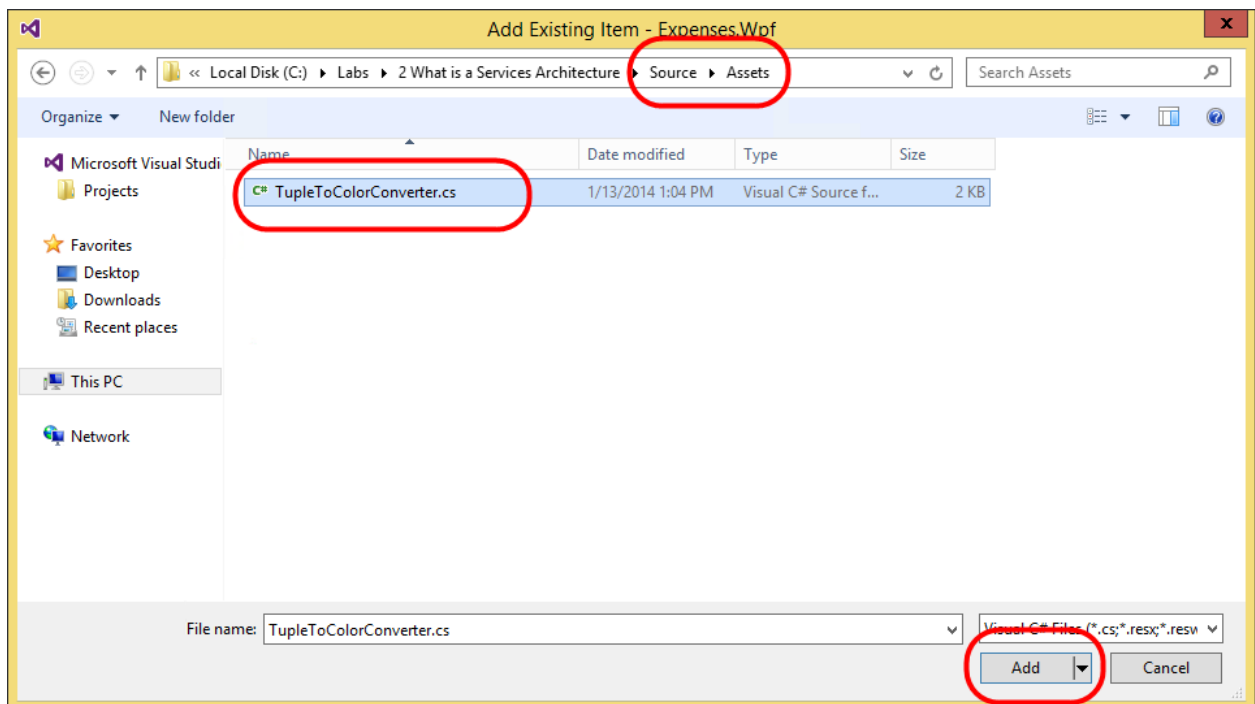
## Task 2: Binding a view to the property

In this task, we'll bind the summary view to use the new property.

1. Right-click the **Converters** folder in the **Expenses.Wpf** project and select **Add | Existing Item....**

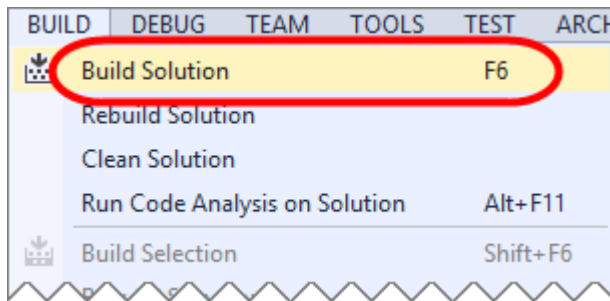


2. Navigate to the supporting files for this lab and add “\Source\Assets\TupleToColorConverter.cs”.

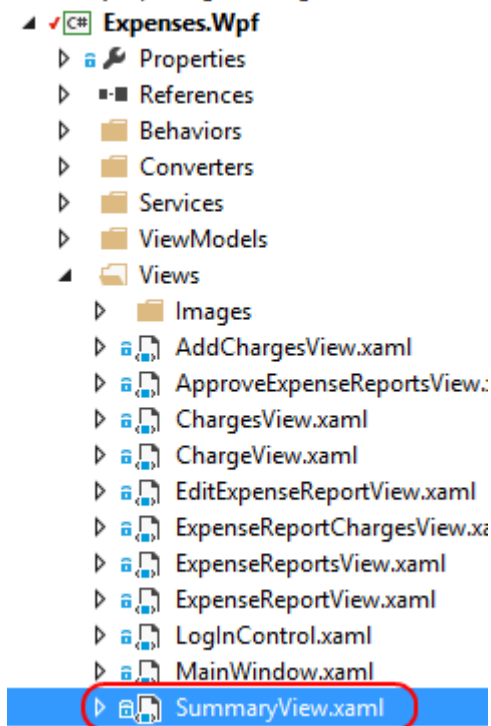


3. In the **Solution Explorer**, double-click **TupleToColorConverter.cs** to open it. This class is a simple converter that accepts a **Tuple<byte, byte, byte>** and returns a **SolidColorBrush**. We'll need it in the next step where we bind the new property to a **Grid's Background**.
4. From the Visual Studio menu bar, select “**BUILD**” | “**Build Solution**”. Ensure the solution builds successfully. Fix any issues that arise. Doing the Build here makes sure that the new **TupleToColorConverter** class we just added is recognized by Visual Studio and the XAML designer. That will make the next steps less error prone.





5. In the **Solution Explorer**, double-click the **SummaryView.xaml** file in the **Views** folder of the **Expenses.Wpf** project to open it.



6. Before we can bind our **Tuple** property to a **Brush** property, we'll need to add our converter as a resource. Add the following line as the first line under the **<UserControl.Resources>** node.

```
<expenses:TupleToColorConverter x:Key="TupleToColorConverter" />
```

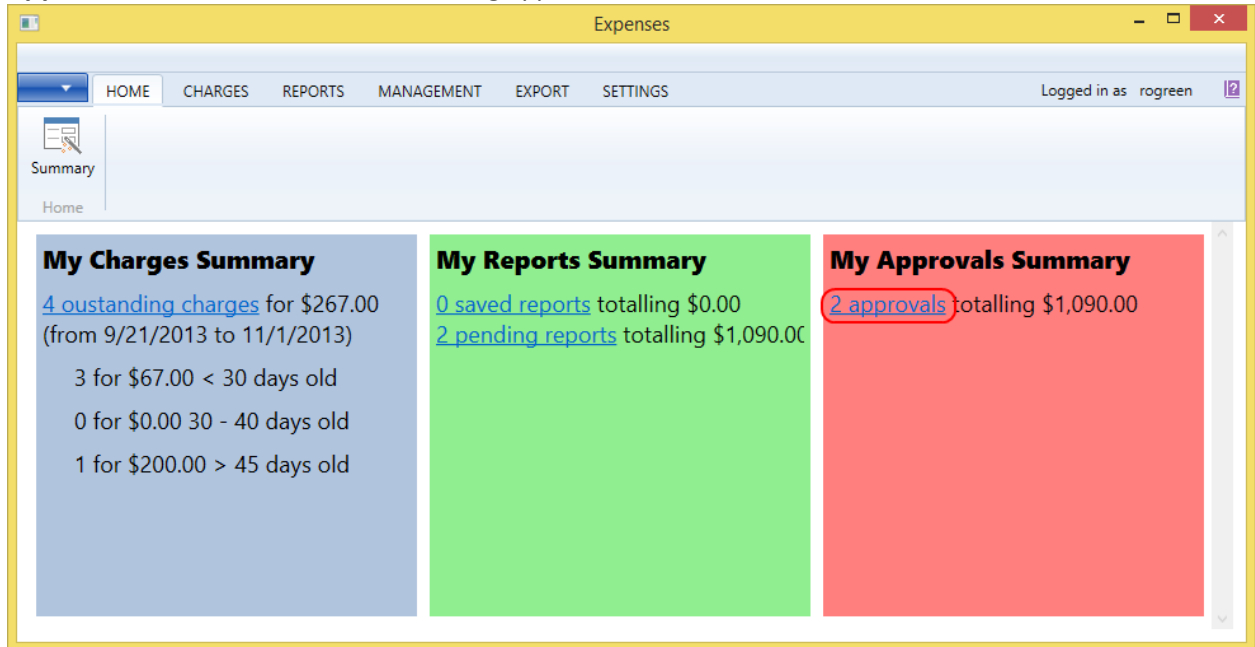
7. Now we can perform the binding itself. At around line 84 there is a **Grid** with the background set to **"LightYellow"**. Change the background value inside the quotes to the following (this is one continuous line without a line break). This tells the **Grid** to use our property as its background color, and that it can use the referenced converter to find a **Brush** from the **Tuple** value.

```
{Binding MyApprovalsBackgroundColor, Converter={StaticResource  
TupleToColorConverter}}
```

### Task 3: Confirming the feature

In this task, we'll build and run the app to confirm our feature works as designed.

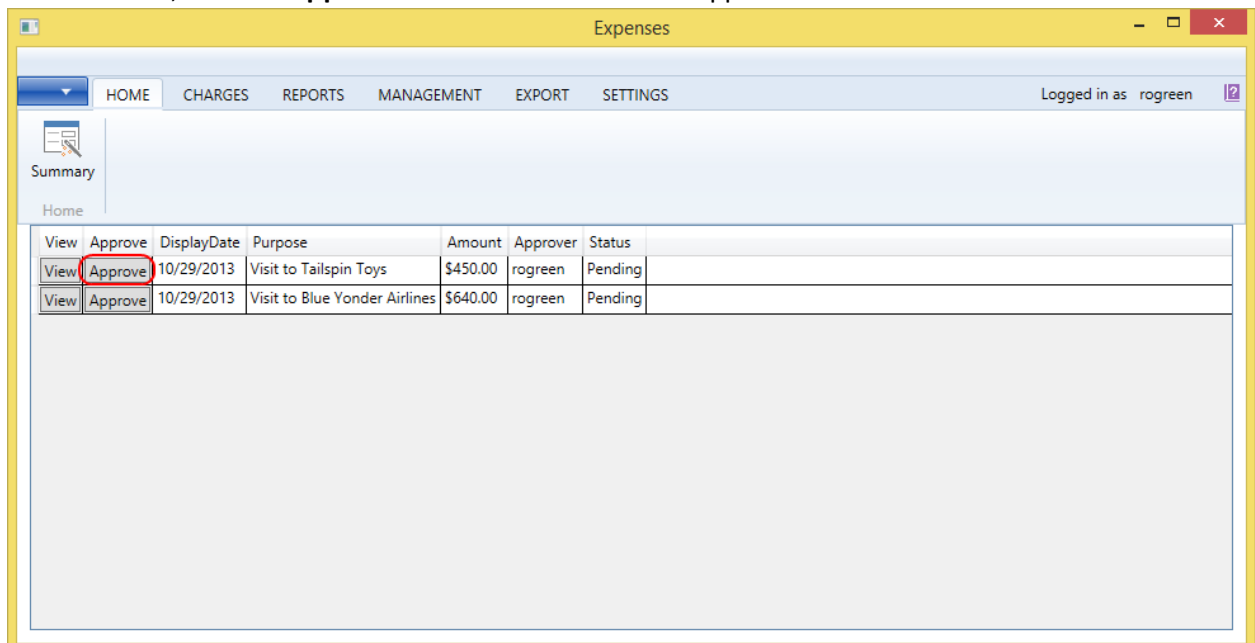
1. Press **F5** to build and run the app.
2. Note that when it loads, the "My Approvals Summary" now has a red background. Click the **2 Approvals** link to view the items awaiting approval.



The screenshot shows the 'Expenses' application interface. The top navigation bar includes 'HOME', 'CHARGES', 'REPORTS', 'MANAGEMENT', 'EXPORT', and 'SETTINGS'. The user is logged in as 'rogreen'. The main content area features three summary cards:

- My Charges Summary** (blue background): 4 outstanding charges for \$267.00 (from 9/21/2013 to 11/1/2013). Details: 3 for \$67.00 < 30 days old, 0 for \$0.00 30 - 40 days old, 1 for \$200.00 > 45 days old.
- My Reports Summary** (green background): 0 saved reports totalling \$0.00, 2 pending reports totalling \$1,090.00.
- My Approvals Summary** (red background): 2 approvals totalling \$1,090.00. The '2 approvals' link is circled in red.

3. For each item, click the **Approve** button and conform the approval.



The screenshot shows the 'Expenses' application interface with a table of pending approvals. The 'Approve' button for the first item is circled in red.

View	Approve	DisplayDate	Purpose	Amount	Approver	Status
<a href="#">View</a>	<a href="#">Approve</a>	10/29/2013	Visit to Tailspin Toys	\$450.00	rogreen	Pending
<a href="#">View</a>	<a href="#">Approve</a>	10/29/2013	Visit to Blue Yonder Airlines	\$640.00	rogreen	Pending

4. From the **Home** tab, click the **Summary** button to view the summary view. Note that the approvals section now has a green background.

The screenshot shows the 'Expenses' application window. The top navigation bar includes 'HOME', 'CHARGES', 'REPORTS', 'MANAGEMENT', 'EXPORT', and 'SETTINGS'. The 'HOME' tab is selected. Below the navigation bar, there is a 'Summary' button with a document icon. The main content area is divided into three sections: 'My Charges Summary' (blue background), 'My Reports Summary' (green background), and 'My Approvals Summary' (green background). The 'My Approvals Summary' section is highlighted with a green background.

Section	Summary
My Charges Summary	4 <a href="#">outstanding charges</a> for \$267.00 (from 9/21/2013 to 11/1/2013) 3 for \$67.00 < 30 days old 0 for \$0.00 30 - 40 days old 1 for \$200.00 > 45 days old
My Reports Summary	0 <a href="#">saved reports</a> totalling \$0.00 2 <a href="#">pending reports</a> totalling \$1,090.00
My Approvals Summary	0 <a href="#">approvals</a> totalling \$0.00