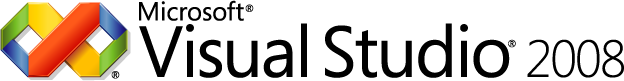
* 1. 

Hands-On Lab

Introduction to ASP.NET MVC

Lab version: 1.0.0

Last updated: 3/13/2009

* 1. 

# Contents

[Overview 4](#_Toc224707066)

[Exercise 1: Creating an ASP.NET MVC Application 7](#_Toc224707067)

[Task 1 – Creating an ASP.NET MVC Web Application Project 7](#_Toc224707068)

[Task 2 – Exploring the Solution Structure 10](#_Toc224707069)

[Task 2 – Understanding the Controllers 12](#_Toc224707070)

[Task 3 – Understanding the Views 15](#_Toc224707071)

[Task 4 – Understanding the ASP.NET URL Routing 16](#_Toc224707072)

[Exercise 2: Developing an ASP.NET MVC Application 19](#_Toc224707073)

[Task 1 – Creating an Entity Data Model 20](#_Toc224707074)

[Task 2 – Implementing Customer View Data 23](#_Toc224707075)

[Task 3 – Implementing Address View Data 25](#_Toc224707076)

[Task 4 – Implementing Customer Controller 27](#_Toc224707077)

[Task 5 – Implementing Address Controller 29](#_Toc224707078)

[Task 6 – Creating Customer Controller Index View 33](#_Toc224707079)

[Task 7 – Creating Customer Controller Info View 35](#_Toc224707080)

[Task 8 – Creating Address Controller Edit View 38](#_Toc224707081)

[Task 9 – Creating Address Controller New View 40](#_Toc224707082)

[Exercise 2: Verification 43](#_Toc224707083)

[Verification 1 43](#_Toc224707084)

[Exercise 3: Testing an ASP.NET MVC Application 50](#_Toc224707085)

[Task 1 – Opening the ASP.NET MVC Application 51](#_Toc224707086)

[Task 2 – Testing Application Routes 51](#_Toc224707087)

[Task 3 – Testing Customer Controller 55](#_Toc224707088)

[Exercise 3: Verification 58](#_Toc224707089)

[Verification 1 58](#_Toc224707090)

[Verification 2 59](#_Toc224707091)

[Summary 60](#_Toc224707092)

[Appendix 61](#_Toc224707093)

[Moq Library 61](#_Toc224707094)

Overview

* 1. The **Model View Controller (MVC)** architectural pattern separates an application into three main components:
  + **Models**: Model objects are the parts of the application that implement the domain logic. Often, model objects also retrieve and store model state in a database.
  + **Views:** Views are the components that display the application's user interface (UI). Typically, this UI is created from the model data. An example would be an edit view of a Products table that displays text boxes, drop-down lists, and check boxes based on the current state of a Product object.
  + **Controllers:** Controllers are the components that handle user interaction, manipulate the model, and ultimately select a view to render the UI. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction.
  1. The MVC pattern helps you to create applications that separate the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements. This separation helps you manage complexity when you build an application, because it enables you to focus on one aspect of the implementation at a time. In addition to managing complexity, the MVC pattern makes it easier to test applications than it is to test a traditional ASP.NET Web application, encouraging the use of test-driven development (TDD) for creating an applications.
  2. Then, the **ASP.NET MVC** framework provides an alternative to the ASP.NET Web-forms pattern for creating MVC-based Web applications. The **ASP.NET MVC** framework is a lightweight, highly testable presentation framework that (as with Web-forms-based applications) is integrated with existing ASP.NET features, such as master pages and membership-based authentication.
  3. In addition, the loose coupling between the three main components of an MVC application also promotes parallel development. For instance, one developer can work on the view, a second developer can work on the controller logic, and a third developer can focus on the business logic in the model.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Understand ASP.NET MVC framework
  + Create an ASP.NET MVC application
  + Perform Testing when creating an ASP.NET MVC application

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2008 SP1
  + Microsoft ASP.NET MVC 1.0
  + Microsoft SQL 2005 or Microsoft SQL 2008 (Express edition or above)
  + Adventure Works sample database
    - For Microsoft SQL 2005: [AdventureWorksLT.msi](http://www.codeplex.com/MSFTDBProdSamples/Release/ProjectReleases.aspx?ReleaseId=4004)
      1. **Note**: The msi just copies the sample database files to your file system; you have to manually attach the database to the SQL Server. For more information, see [How to: Attach a Database (SQL Server Management Studio)](http://technet.microsoft.com/en-us/library/ms190209.aspx).
      2. *Tip for Windows Vista Users*: Consider installing the database files to other location than "C:\Program Files\", as this this folder has extra security and you might not be able to save changes in project files as they will be read-only.
    - For Microsoft SQL 2008: [AdventureWorks 2008 sample databases](http://www.codeplex.com/MSFTDBProdSamples/Release/ProjectReleases.aspx?ReleaseId=18407)
      1. **Note:** The msi will automatically install all the sample databases in your SQL Server. However, you will only use **AdventureWorksLT** DB.

# Setup

* 1. All the requisites for this lab are verified using the Dependency Checker. To make sure that everything is correctly configured, follow these steps:
  2. **Note:** To perform the setup steps you need to run the scripts in a command window with administrator privileges.
  3. Run the Dependency checker for the Training Kit if you haven't done it previously. To do this, run the **CheckDependencies.cmd** script located under the **%TrainingKitInstallationFolder%\Labs\aspNetMvc\Setup** folder. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.
     1. **Note:** For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **CheckDependencies.cmd** file launches the Visual Studio installer file that installs the code snippets.

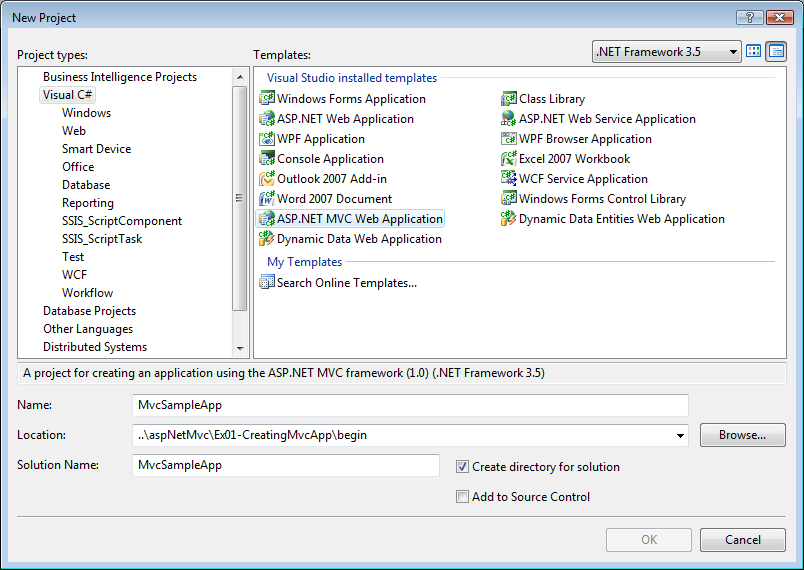
Exercises

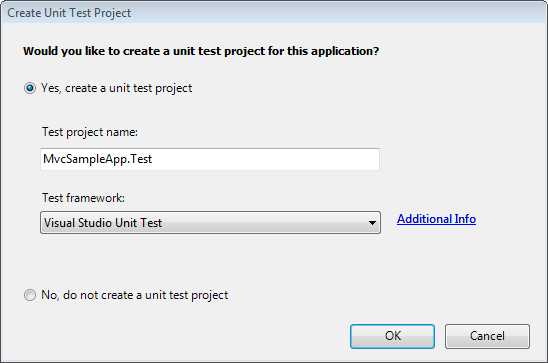
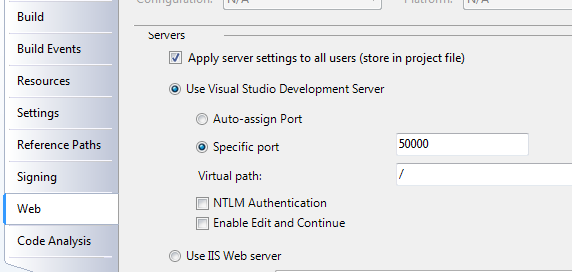
* 1. This Hands-On Lab is comprised by the following exercises:
  2. Creating an ASP.NET MVC Application
  3. Developing an ASP.NET MVC Application
  4. Testing an ASP.NET MVC Application
  5. Estimated time to complete this lab: **90 minutes**.
  6. **Note:** Each exercise is accompanied by a starting solution. Some code sections are missing from these solutions, which will be completed through each exercise. Therefore the starting solutions will not work if you run them directly.
  7. Inside each exercise you will also find an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Exercise 1: Creating an ASP.NET MVC Application

1. In this exercise you will learn how to create an ASP.NET MVC application in Visual Studio and introduce the default project structure and conventions.
   1. The ASP.NET MVC framework separates the model, view, and controller components. The model component typically maintains state by persisting data in a database. The view component is selected by the controller and renders the appropriate UI. By default, the ASP.NET MVC framework uses the existing ASP.NET page (.aspx), master page (.master), and user control (.ascx) types for rendering to the browser. The controller component locates the appropriate action method in the controller, gets values to use as the action method's arguments, and handles any errors that might occur when the action method runs. It then renders the requested view. By default, each set of components is in a separate folder of an MVC Web application project.

Task 1 – Creating an ASP.NET MVC Web Application Project

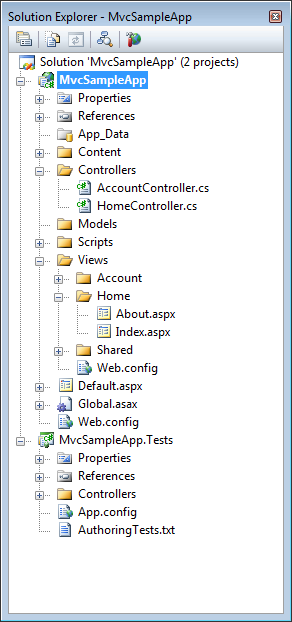
* 1. In this task you will create and configure an empty ASP.NET MVC application project using the MVC Visual Studio template.
  2. Open Microsoft Visual Studio 2008 from **Start** | **All Programs** | **Microsoft Visual Studio 2008** | **Microsoft Visual Studio 2008**.
  3. On the **File** menu, point to **New**, and click **Project**.
  4. In the **New Project** dialog box make sure that **.NET Framework 3.5** is selected and select the **Visual C# | ASP.NET MVC Web Application** project type. You may set the location to **AspNetMvc\Ex01-CreatingMvcApp\begin** which is the provided folder.
  5. Change the **Name** to **MvcSampleApp** and click **OK**.
     1. 
     2. Figure
     3. Create New Project Dialog Box

1. After selecting the **OK** button, you’ll be asked whether you’d like to create a test project as well. Select **Yes**, enter **MvcSampleApp.Test** as the name of the project, and then click **OK**.
   * 1. **Note:** When you create a new MVC Web application, Visual Studio gives you the option to create two projects at the same time. The first project is a Web project where you can implement your application. The second project is a testing project where you can write unit tests for your MVC components.
     2. 
     3. Figure
     4. Create Unit Tests Dialog Box
     5. **Note:** The **Test framework** drop-down list on the test project dialog window currently only has an option for **Visual Studio Unit Test**. This list is extensible and will include other testing framework options when they’re installed on your machine. This will enable you to easily begin writing unit tests against your ASP.NET MVC application using your favorite unit testing framework.
   1. Configure the web site to use port 50000. This step is needed for consistency with the end solution provided.
      1. To do this, in **Solution Explorer**, right-click **MvcSampleApp** project and in the context menu select **Properties**.
      2. In the **Property** pages open the **Web** tab.
      3. In the **Servers** section select **Specific Port**.
      4. Set the port number to **50000**.
      5. Press **Ctrl** + **S** to save changes.
         1. 
         2. Figure
         3. Specifying a port number

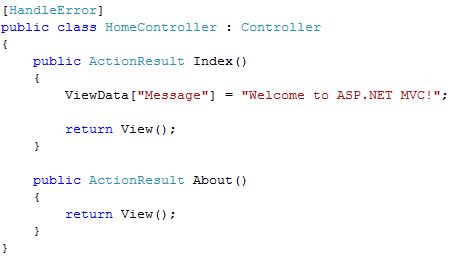
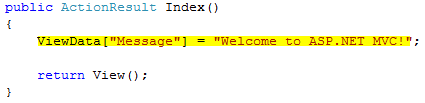
Task 2 – Exploring the Solution Structure

The ASP.NET MVC framework includes a Visual Studio project template that helps you create Web applications that are structured to support the MVC pattern. This template creates a new MVC Web application that is configured to have the required folders, item templates, and configuration-file entries.

In this task you will examine the solution structure to understand the involved elements and its relationships.

* 1. Press **Ctrl+ Alt + L** to see the **Solution Explorer** and expand the folders to expose its content.
     1. 
     2. Figure
     3. ASP.NET MVC Solution Structure
     4. When you create an ASP.NET MVC Web application project, MVC components are separated based on the following project folders:
     + **App\_data:** The App\_Data folder is the physical store for data. This folder has the same role as it does in ASP.NET Web sites that use Web Forms pages.
     + **Content**: The Content folder is the recommended location to add content files such as cascading style sheet files, images, and so on. In general, the Content folder is for static files.
     + **Controllers:** Controller classes. In a MVC based application are the components responsible for handling end user interaction, manipulating the model, and ultimately choosing a view to render to display UI.
       1. **Note:** The MVC framework requires the names of all controllers to end with "Controller"—for example, HomeController, LoginController, or ProductController.
     + **Models**: The Models folder is provided for classes that represent the application model for your MVC Web application. This usually includes code that defines objects and that defines the logic for interaction with the data store. Typically, the actual model objects will be in separate class libraries. However, when you create a new application, you might put classes here and then move them into separate class libraries at a later point in the development cycle.
     + **Scripts**: This is the recommended place to store JavaScript files in your application.
     + **Views**: The Views folder is the recommended location for views. Views are the components responsible for displaying the application's user interface. Views use .aspx, .ascx, and .master files, in addition to any other files that are related to rendering views. The Views folder contains a folder for each controller; the folder is named with the controller-name prefix. For example, if you have a controller named **HomeController**, the Views folder will contain a folder named Home. By default, when the ASP.NET MVC framework loads a view, it looks for an .aspx file that has the requested view name in the Views\controllerName folder (**Views\[ControllerName]\[Action].aspx**).
     + **Views\Shared:** By default, there is also a folder named Shared in the Views folder, which does not correspond to any controller. The Shared folder is used for views that are shared across multiple controllers. For example, you can put the Web application's master page in the Shared folder.
     1. **Note:** In addition to the folders listed previously, an MVC Web application uses the **Global.asax** file to set global URL routing defaults, and it uses the Web.config file to configure the application.

Task 2 – Understanding the Controllers

* 1. In ASP.NET applications that do not use the MVC framework, user interaction is organized around pages, and around raising and handling events from those pages. In contrast, user interaction with ASP.NET MVC applications is organized around controllers and their action methods.
  2. The ASP.NET MVC framework maps URLs to classes that are referred to as controllers. Controllers process incoming requests, handle user input and interactions, and execute appropriate application logic. A controller class typically calls a separate view component to generate the HTML markup for the request. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction.
  3. Open the **Home Controller**. To do this, in the Solution Explorer, double-click the **HomeController.cs** file under **Controllers** folder. You will see the following code:
     1. **Note:** The default behavior of the MVC framework requires that all controller classes must be suffixed with "**Controller**". This convention can be modified if desired. All controller classes must implement the **IController** interface (or inherit from the **Controller** base class, which in turn implements **IController**).
     2. The controller defines action methods. In the typical workflow of an MVC Web application, controller action methods handle the incoming web request. These action methods use the incoming parameter values to execute application code, retrieve or update data model objects from a database, and select a view that renders a response to a browser.
     3. 
     4. Figure
     5. Viewing the Controller’s auto generated code
     6. **Note:** The **[HandleError]** attribute filter provides a way to declaratively indicate on a Controller or Action method that a friendly error response should be displayed if an error occurs during the processing of an ASP.NET MVC request.
     7. **Note:** Notice that the methods name binds to actions in the request URL.   
        In order for an action method to be callable, it must be public, and not have a **[NonActionAttribute]** attached to it.
     8. Action methods must return an **ActionResult** instance. An action result is what a controller action returns after executing, in response to a browser request. This can include: rendering a view, redirecting to another action, redirecting to another page, etc.
     9. ASP.NET MVC framework supports several types of action results including:
     10. - **ViewResult**: Represents HTML and markup.
     11. - **EmptyResult:** Represents no result.
     12. - **RedirectResult**: Represents a redirection to a new URL.
     13. - **JsonResult**: Represents a JavaScript Object Notation result that can be used in an AJAX application.
     14. - **JavaScriptResult**: Represents a JavaScript script.
     15. - **ContentResult**: Represents a text result.
     16. - **FileContentResult**: Represents a downloadable file (with the binary content).
     17. - **FilePathResult**: Represents a downloadable file (with a path).
     18. - **FileStreamResult**: Represents a downloadable file (with a file stream).
     19. In this case the actions do not return a **ViewResult()**,but the **View()** method of the **Controller** base class. Normally, you do not return an **ActionResult** directly; instead, you call one of the following methods of the **Controller** base class:
     20. - **View**: Returns a **ViewResult** action result.
     21. - **Redirect**: Returns a **RedirectResult** action result.
     22. - **RedirectToAction**: Returns a **RedirectToRouteResult** action result (redirects to the specified action).
     23. - **RedirectToRoute**: Returns a **RedirectToRouteResult** action result (redirects to the specified route).
     24. - **Json**: Returns a **JsonResult** action result.
     25. - **JavaScriptResult**: Returns a **JavaScriptResult**.
     26. - **Content**: Returns a **ContentResult** action result.
     27. - **File**: Returns a **FileContentResult**, **FilePathResult**, or **FileStreamResult** depending on the parameters passed to the method.
     28. For more information, see <http://www.asp.net/learn/mvc/tutorial-03-cs.aspx>.
  4. Views should only render their output using the view-specific data provided by the Controller class. In the ASP.NET MVC Framework we call this view-specific data, **ViewData**.
     1. **Note:** To pass data to the view, you can use the **ViewData** property of the **ControllerBase** class. This property returns a **ViewDataDictionary** object that has case-insensitive string keys. You can use the **ViewData** dictionary for this or use strongly-typed data as we will cover in this lab. Notice that in the **Home** controller, the methods set values in the **ViewData** dictionary and returns to the view associated to the action calling the **View** method of the controller.
     2. 
     3. Figure
     4. Using the ViewData dictionary

Task 3 – Understanding the Views

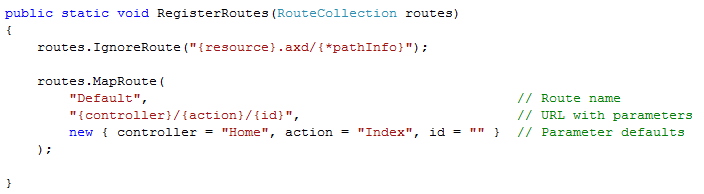
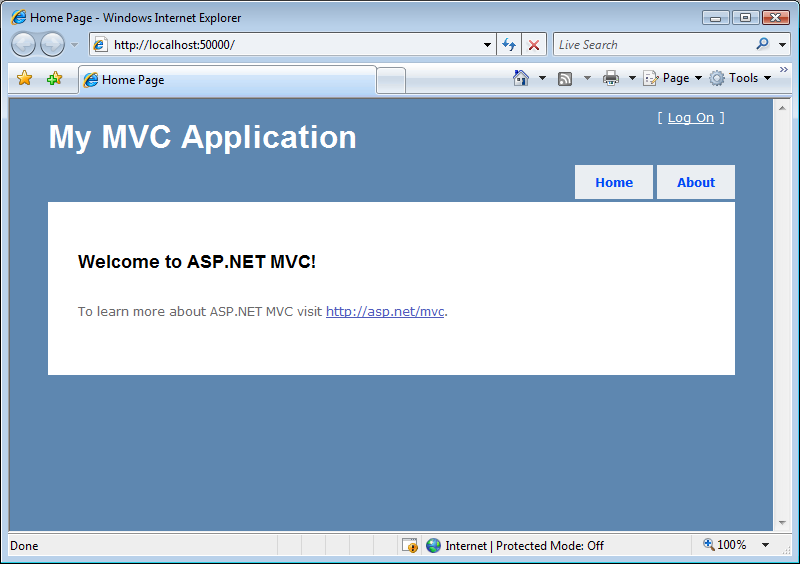
* 1. In the typical workflow of an MVC Web application, controller action methods handle an incoming Web request. These action methods use the incoming parameter values to execute application code, and retrieve or update data model objects from a database. They then select a view that renders a response to a browser.

In an MVC application the views are the components responsible for displaying the application's user interface. Views are intended exclusively for encapsulating presentation logic. They should not contain any application logic or database retrieval code. Views render the appropriate UI by using this View Data class which is a MVC view-related data object that the controller provides when it calls the method to render the view.

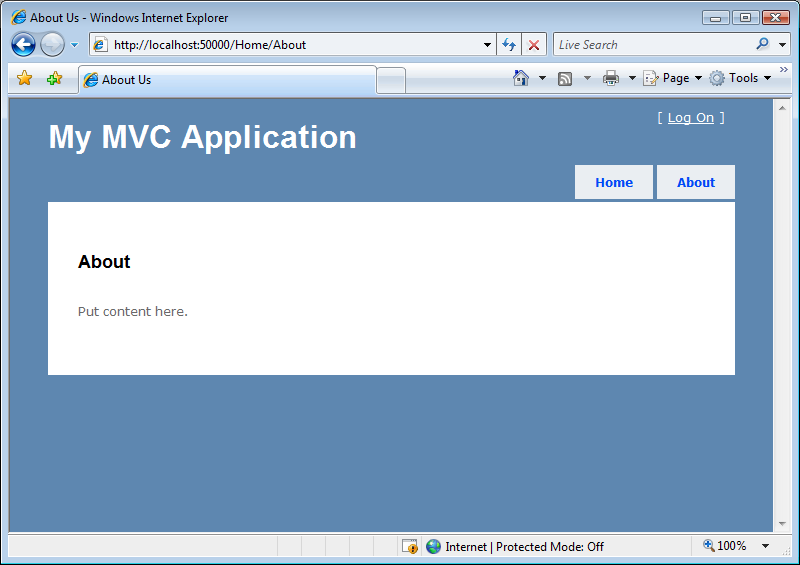
The views use .aspx, .ascx, and .master files, as well as any other files that are related to rendering views. The Views folder contains a folder for each controller that is named with the controller prefix.

* 1. Open the **Index** view of the **Home** controller. To do this, in **Solution Explorer**, double-click the **Index.aspx** file under **Views\Home** folder.
     1. **Note:** View-templates do not have a code-behind file by default. This is mainly to reinforce the purpose of views in a MVC application which are intended to be purely about rendering and to not contain any non-rendering related code.
     2. 
     3. Figure
     4. Viewing the View’s auto generated code
     5. **Note:** The **Html** object is an instance of the **HtmlHelper** class which provides useful common methods to generate HTML tags. The **Encode** method applies HTML encoding to a specified string. In this case, the View is encoding the **Message** value received from the Controller through the **ViewData** dictionary.
     6. From within your View you can access the **ViewData** in either a late-bound or strongly-typed way. If your View derives from **ViewPage**, the **ViewData** property will be typed as a late-bound dictionary. If your View derives from the generics based **ViewPage<T>**, where **T** indicates the data object type of the **ViewData** the controller is passing to the View, the **ViewData** property will be strongly typed to match the same type that your controller passed in.

Task 4 – Understanding the ASP.NET URL Routing

* 1. The ASP.NET MVC framework uses **ASP.NET Routing** to map URLs to controller classes and actions. **ASP.NET Routing** parses variables in the URL according to a pattern that you define, and automatically passes the variables to a controller action as parameter arguments. In this way the URLs do not have to map to specific files in a Web site.
  2. By default ASP.NET MVC projects have a preconfigured set of URL routing rules that enable you to easily get started on an application without needing to explicitly configure anything. You can start developing using a default set of name-based URL mapping conventions that are declared within the ASP.NET **Application** class of the **Global.asax** file created by the new ASP.NET MVC project template in Visual Studio.
  3. The preconfigured routing rule indicates that the ASP.NET MVC framework should by default map URLs to Controllers using a ***[controller]/[action]/[id]*** pattern when determining which **Controller** class to instantiate, and which **Action** method to invoke (along with which parameters should be passed in).
  4. **Note:** Double-click **Global.asax** in **Solution Explorer** to see how the preconfigured routes are defined.
  5. Start a new instance of the **MvcSampleApp** project. To do this, in **Solution Explorer** right-click **MvcSampleApp** project, point to **Debug** and select **Start New Instance**.
     1. **Note:** If the dialog **Debugging Not Enabled** appears, select **Modify the Web.config file to enable debugging** and click **OK**.
     2. A request will be made to [**http://localhost:50000**](http://localhost:50000) which will be intercepted by the **ASP.NET Routing** engine applying the ***Default*** registered route (the pattern is ***[controller]/[action]/[id]****)*. Since the URL does not contain any controller to map, the routing engine will instantiate a default controller (*Home*), and invoke a default action (*Index*) specified in the registered route.
     3. 
     4. Figure
     5. Viewing the default registered route in Global.asax
     6. In this case, the default parameters for the route are **Home** as the controller, and **Index** as the action; that’s why **Home** controller’s **Index** View is rendered.
     7. 
     8. Figure
     9. MVC Sample Application Home Page

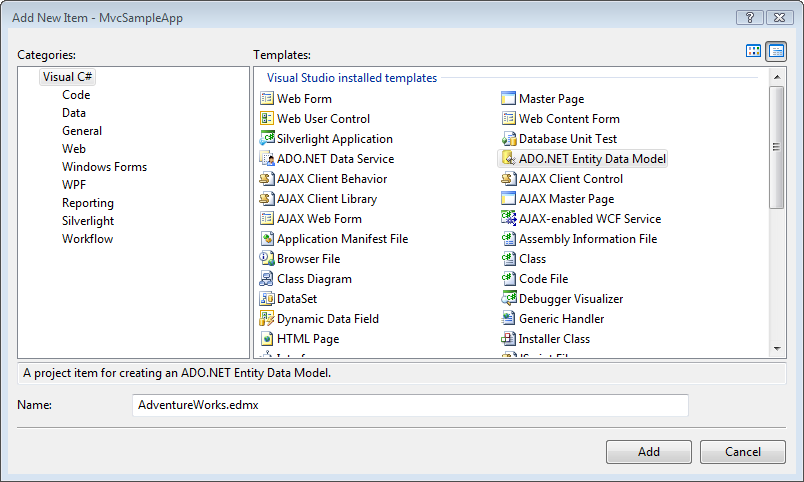
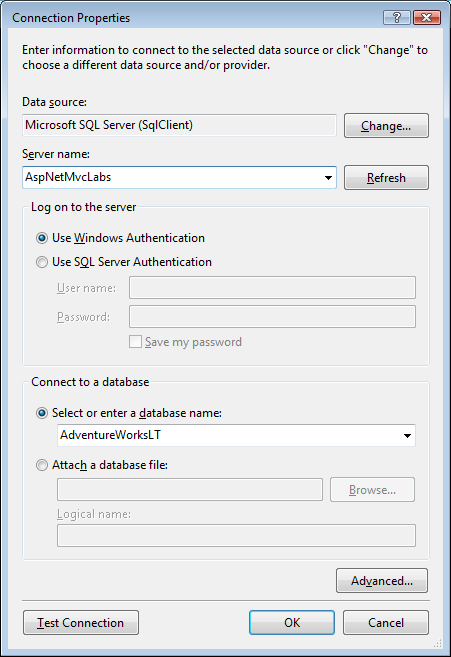
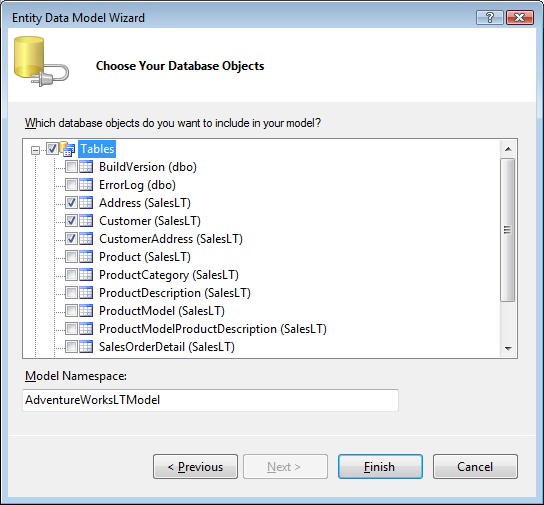
**Note:** The default URL routing pattern is as indicated before: **…/[ControllerName]/[Action]/[Parameters]**. You can define new routing rules in the **RegisterRoutes** method in **Global.asax.cs** file. Routes are initialized in the **Application\_Start** method of the **Global.asax.cs** or **Global.asax.vb** file.

* 1. Browse to the **About** Page by clicking **About** link on page header. You will be redirected to the following address in the web browser <http://localhost:50000/Home/About> which invokes **About** method on the **Home Controller**.
     1. 
     2. Figure
     3. About Page

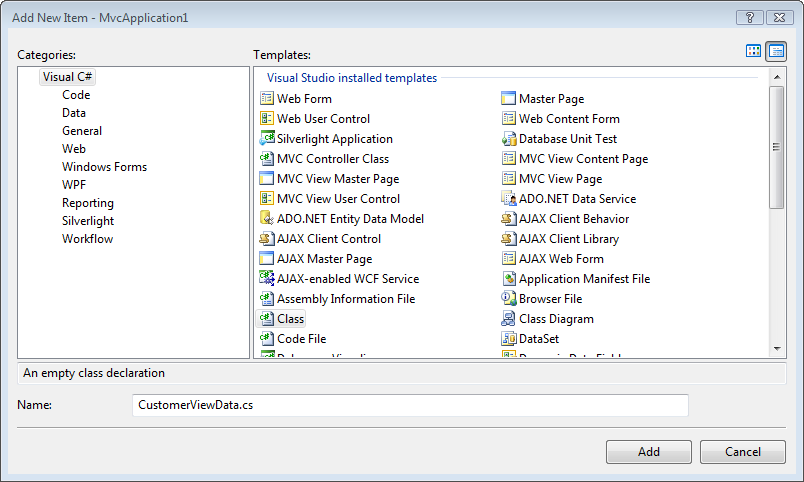
Exercise 2: Developing an ASP.NET MVC Application

* 1. In this exercise you will learn how to develop an ASP.NET MVC application by going through the process of creating controllers, views and models.
  2. You will build an ASP.NET MVC application that display a paged list of customers, showing their information and allowing to create, edit and delete the customer’s addresess. The application will have three controllers: the customer controller which handles customer listing and customer information, the address controller to handle the edition, creation and deletion of addresses and finally the home controller to handle application welcome views.
  3. **Note:** To verify that each step is correctly performed, it is recommended to build the solution at the end of each task.

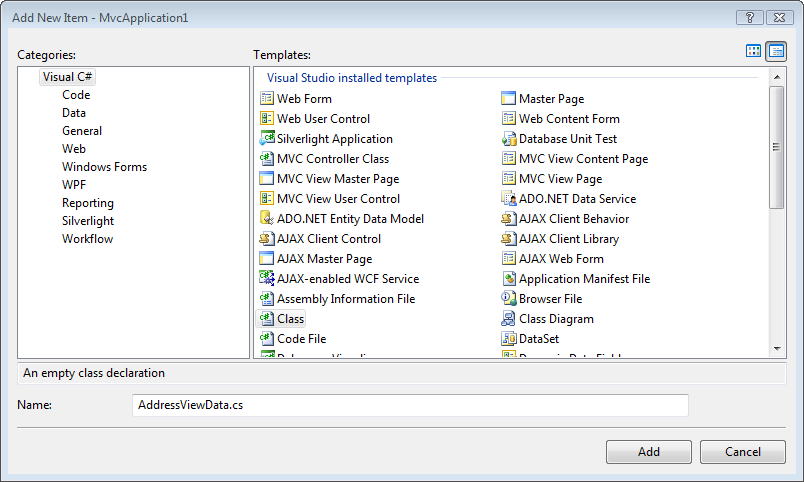
Task 1 – Creating an Entity Data Model

* 1. In this task you will create the mapping specification that connects programmable classes to storage structures using an **Entity Data Model** (EDM) which is a specification for defining the data used by applications built on the Entity Framework.
  2. Open Microsoft Visual Studio 2008 from **Start | All Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008**.
  3. Open the solution file **MvcSampleApp.sln** located under **AspNetMvc\Ex02-DevelopingMvcApp\begin\**. Alternatively, you may continue working with the solution obtained after completing the previous exercise.
  4. In **Solution Explorer**, open the **Shared** folder under **Views** folder. To do this, click the plus sign next to the folders names. Select the **Site.Master** file. Right-click the file and select **Delete**.
  5. Import the provided **Site.Master**. To do this, right-click the **Shared** folder, point to **Add** and select **Existing Item**. On the **Add Existing Item** dialog browse to **AspNetMvc\Assets\Shared** and select all files in the folder. Click **Add**.
     1. **Note:** Like traditional ASP.NET Web pages, ASP.NET page views (.aspx files) can use master pages which provide the ability to define common structure and interface elements for the site.
  6. Create the **AdventureWorks** Entity Data Model. To do this, in **Solution Explorer**, right-click the **Models** folder in **MvcSampleApp** project, point to **Add**, and click **New Item**.
  7. In the **Add New Item** dialog box select **ADO.NET Entity Data Model.** Specify a Name value of **AdventureWorks.edmx**, and then click **Add**.
     1. 
     2. Figure
     3. Adding the ADO.NET Entity Data Model
  8. After the **Entity Data Model Wizard** opens, select **Generate From Database** and click **Next**.
  9. Specify the Database connection. To do this, click **New Connection**.
  10. In the **Choose Data Source** dialog, select **Microsoft SQL Server** as **Data Source** and click **Continue**.
  11. In the **Connection Properties** dialog window, enter **AspNetMvcLabs** as **Server Name**, then select **AdventureWorksLT** database and click **OK**.
      1. **Note:** **AspNetMvcLabs** is the default alias for the database server installed when the **SetupEx.cmd** script was run at the beginning of this lab.
      2. 
      3. Figure
      4. Specify the database connection
  12. Back on the Entity Data Model wizard click **Next**.
  13. Include only the following tables from all the proposed Database objects:
      + **Address (SalesLT)**
      + **Customer (SalesLT)**
      + **CustomerAddress (SalesLT)**
  14. Leave the **Model Namespace** by defaultand click **Finish**.
      1. 
      2. Figure
      3. Choose the Database Objects to Include in the Model
  15. Add the **AdventureWorksRepository**. The repository exposes methods to retrieve entities from the model generating a level of abstraction from the underlying data model. Right-click the **Models** folder of **MvcSampleApp** project in **Solution Explorer**, point to **Add** and click **Existing Item**. Browse to **AspNetMvc\Assets\**, select **AdventureWorksRepository.cs** and click **Add**.
      1. **Note:** For information see [ADO.NET Entity Framework](http://msdn2.microsoft.com/en-us/library/bb387122.aspx).

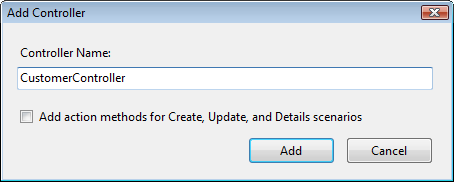
Task 2 – Implementing Customer View Data

* 1. In this task you will create the View Data that will be used in the customer controller to display the paged list of customers.
  2. Create the **ViewData** folder. To do this, right-click **MvcSampleApp** project, point to **Add** and select **New Folder**. Set **ViewData** as the folder name.
  3. Create the **CustomerViewData** class. Right-click the **ViewData** folder, point to **Add** and select **New Item**.
  4. In the **Add New Item** dialog box select **Class.** Specify a Name value of **CustomerViewData.cs**, and then click **Add**.
     1. 
     2. Figure
     3. Adding CustomerViewData class
  5. In **CustomerViewData.cs**, replace all the using statements created by default with the following code.
     1. C#
     2. **using System;**
     3. **using System.Web;**
     4. **using System.Web.Mvc;**
     5. **using System.Collections.Generic;**
     6. **using MvcSampleApp.Models;**
  6. Add the following code (**bolded**) to the **CustomerViewData** class to implement its properties. The class provides the view a collection of customers and the numbers of the next and previous pages to generate links.
     1. (Code Snippet – *Intro to Asp.Net MVC Lab - CustomerViewData Properties*)
     2. C#
     3. public class CustomerViewData
     4. {
     5. **public IEnumerable<Customer> Customers**
     6. **{**
     7. **get;**
     8. **set;**
     9. **}**
     10. **public int PreviousPage**
     11. **{**
     12. **get;**
     13. **set;**
     14. **}**
     15. **public int NextPage**
     16. **{**
     17. **get;**
     18. **set;**
     19. **}**
     20. }

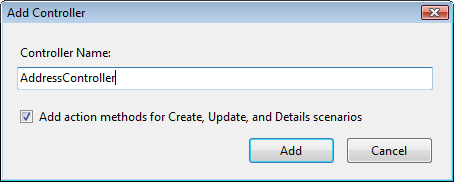
Task 3 – Implementing Address View Data

* 1. In this task you will create the view data that will be used in the address controller to allow the editing of an address.
  2. Create the **AddressViewData** class. Right-click the **ViewData** folder, point to **Add** and select **New Item**.
  3. In the **Add New Item** dialog box select **Class.** Specify a Name value of **AddressViewData.cs**, and then click **Add**.
     1. 
     2. Figure
     3. Adding AddressViewData class
  4. In **AddressViewData.cs**, replace all the namespace directives created by default with the following code. If the file doesn’t open by default in **Solution Explorer** double-click on **AddressViewData.cs** under the **ViewData** folder.
     1. C#
     2. **using System;**
     3. **using System.Web;**
     4. **using System.Web.Mvc;**
     5. **using MvcSampleApp.Models;**
  5. Add the following code to the **AddressViewData** class to implement its properties. The class provides the view an **Address** entity object and an integer with the id of the customer whose address is being edited.
     1. (Code Snippet *- Intro to Asp.Net MVC Lab – AddressViewData Properties*)
     2. C#
     3. public class AddressViewData
     4. {
     5. **public Address Address**
     6. **{**
     7. **get;**
     8. **set;**
     9. **}**
     10. **public int CustomerId**
     11. **{**
     12. **get;**
     13. **set;**
     14. **}**
     15. }

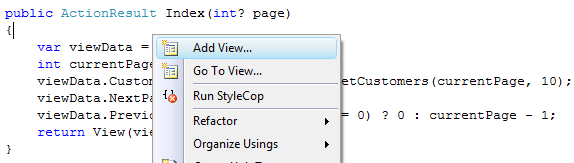
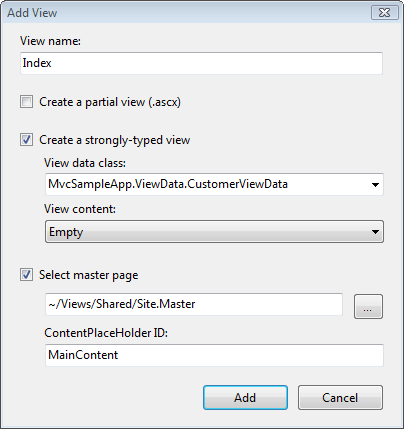
Task 4 – Implementing Customer Controller

* 1. In this task you will create the **Customers** controller with two action methods, one to handle the rendering of a view with a list of customer (**Index** view) and the other to handle customer’s information view (**Info** view).
  2. Create an empty MVC controller class. In **Solution Explorer** right-click the **Controllers** folder, point to **Add** and select **Controller**.
  3. Specify a Controller Name value of **CustomerController**, and then click **Add**.
     1. 
     2. Figure
     3. Adding CustomerController controller
  4. In **CustomerController.cs**, add the following namespace directives:
     1. C#
     2. **using MvcSampleApp.Models;**
     3. **using MvcSampleApp.ViewData;**
  5. Instantiate a repository to act as a data access service. To do this, paste the following code inside the **CustomerController** class.
     1. C#
     2. **private AdventureWorksRepository repository = new AdventureWorksRepository();**
  6. Implement the action method to handle the **CustomerController** **Index** view. To do this, replace the default **Index** method with the following code.
     1. **Note:** The action method creates the view data and fills it with the retrieved list of customers (using paging) and the previous and next page numbers. Finally, it calls the **View** method, passing the view data. Because the name of the view you wish to render has the same name as the action method being executed (i.e. Index), you don’t need to specify the view name in the call to **View***.*
     2. (Code Snippet – *Intro to* *Asp.Net MVC Lab - CustomerController Index Action*)
     3. C#
     4. **public ActionResult Index(int? page)**
     5. **{**
     6. **var viewData = new CustomerViewData();**
     7. **int currentPage = page ?? 0;**
     8. **viewData.Customers = this.repository.GetCustomers(currentPage, 10);**
     9. **viewData.NextPage = currentPage + 1;**
     10. **viewData.PreviousPage = (currentPage <= 0) ? 0 : currentPage - 1;**
     11. **return View(viewData);**
     12. **}**
     13. **Note:** The ASP.NET MVC framework can automatically map URL parameter values to parameter values for action methods. By default, if an action method takes a parameter, the MVC framework examines incoming request data and determines whether the request contains an HTTP request value with the same name. If so, the request value is automatically passed to the action method.
     14. In this case the page number parameter is an optional parameter. Optional parameters in the MVC framework are handled using **nullable** type arguments on controller action methods (For more information, see [Using Nullable Types](http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx)). The MVC framework will either pass in a value if a page number is present in the URL - or pass in null if not.
  7. Implement the action method to handle the **CustomerController** **Info** view. To do this, insert the following code in the **CustomerController** class. This method retrieves the customer object based on the customer id parameter and renders the customer information view.
     1. (Code Snippet – *Intro to* *Asp.Net MVC Lab - CustomerController Info Action*)
     2. C#
     3. **public ActionResult Info(int id)**
     4. **{**
     5. **var customer = this.repository.GetCustomerById(id);**
     6. **return View(customer);**
     7. **}**

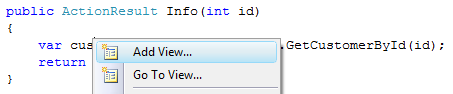
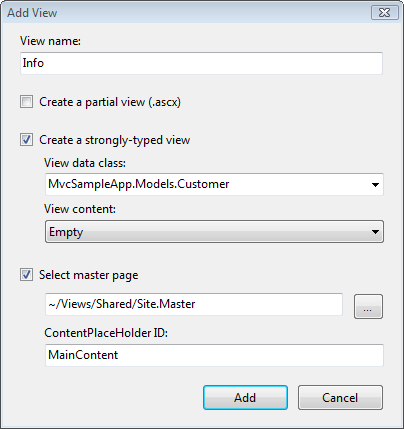
Task 5 – Implementing Address Controller

* 1. In this task you will implement the **Address** controller which handles the edition, creation and deletion of customers addresses. The controller is in charge of rendering two views: the **Edit** view, to edit a customer address and the **New** view to add a new address to a customer. It is also resposible of handling the form submissions sent back from these two views.
  2. Create an empty MVC controller class. In **Solution Explorer** right-click the **Controllers** folder, point to **Add** and select **Controller**.
  3. Specify a Controller Name value of **AddressController**. Make sure the **Add action methods for Create, Update, and Details scenarios** box is checked, and then click **Add**.
     1. 
     2. Figure
     3. Adding AddressController controller
  4. In **AddressController.cs**, add the necessary namespace references (to the data model, view data, etc.). To do this, add the following namespace directives. If the file doesn’t open by default in **Solution Explorer** double-click on **AddressController.cs** under the **Controllers** folder.
     1. C#
     2. **using MvcSampleApp.Models;**
     3. **using MvcSampleApp.ViewData;**
  5. Instantiate a repository to act as a data access service. To do this, paste the following code inside the **AddressController** class.
     1. C#
     2. public class AddressController : Controller
     3. {
     4. **private AdventureWorksRepository repository = new AdventureWorksRepository();**
     5. ...
     6. }
  6. Remove the **Index** and **Details** methods created by default.
  7. Implement the action method to handle the rendering of the **Create** view. To do this, replace the **Create** method for the **GET** operation (not decorated with the **AcceptVerbs(HttpVerbs.Post)** attribute) with the following code:
     1. **Note:** This method receives the **Customer Id** as a parameter, retrieves through the repository the **Customer** entity and finally renders the **Create** view using the **View** method.
     2. (Code Snippet – *Intro to* *Asp.Net MVC Lab – AddressController New Action* )
     3. C#
     4. **public ActionResult Create(int customerId)**
     5. **{**
     6. **AddressViewData addressViewData = new AddressViewData()**
     7. **{**
     8. **CustomerId = customerId**
     9. **};**
     10. **return View(addressViewData);**
     11. **}**
  8. Implement the action method to handle the form submissions of the **Create** view sent back from the browser. To do this, replace the **Create** method for the **POST** operation (decorated with the **AcceptVerbs(HttpVerbs.Post)** attribute) with the following code:
     1. **Note:** This method receives the **customer Id** and a **FormCollection** as a parameter; creates a new **Address** entity object and initializes it using the form submitted from the browser (**UpdateModel** method). Then add the address to the database using the repository and returns to the customer information view using the *RedirectToAction* method.
     2. (Code Snippet – *Intro to* *Asp.Net MVC Lab – AddressController Create Action* )
     3. C#
     4. [AcceptVerbs(HttpVerbs.Post)]
     5. **public ActionResult Create(int customerId, FormCollection collection)**
     6. **{**
     7. **try**
     8. **{**
     9. **AddressViewData addressViewData = new AddressViewData();**
     10. **UpdateModel(addressViewData);**
     11. **this.repository.AddAddress(addressViewData.Address, customerId);**
     12. **return RedirectToAction("Info", "Customer", new { id = customerId });**
     13. **}**
     14. **catch**
     15. **{**
     16. **return View();**
     17. **}**
     18. **}**
     19. **Note:** The **UpdateModel** method is a helper for data binding provided by the ASP.NET MVC Framework. It populates custom classes from form values on Views performing a property assignment for any key that matches a public property on the object.
     20. The **Controller.RedirectToAction** helper method can be used within controllers to return a **RedirectToRouteResult** that will perform redirects to other actions on the same or another controller. The URLs are generated using the URL routing engine.
  9. Implement the action method to handle the rendering of the **Edit** view. To do this, replace the **Edit** method for the **GET** operation (not decorated with the **AcceptVerbs(HttpVerbs.Post)** attribute) with the following code:
     1. **Note:** This method retrieves through the repository the **Address** entity that corresponds to the **Address Id** received as parameter, and then initializes the address view data. Finally, it renders the **Edit** view using the **View** method, and sends the **ViewData** to the view.
     2. (Code Snippet *–* *Intro to* *Asp.Net MVC Lab – AddressController Edit Action*)
     3. C#
     4. **public ActionResult Edit(int addressId, int customerId)**
     5. **{**
     6. **AddressViewData addressViewData = new AddressViewData();**
     7. **addressViewData.Address = this.repository.GetAddressById(addressId);**
     8. **addressViewData.CustomerId = customerId;**
     9. **return View(addressViewData);**
     10. **}**
  10. Implement the action method to handle the form submissions of the **Edit** view sent back from the browser. To do this, replace the **Edit** method for the **POST** operation (decorated with the **AcceptVerbs(HttpVerbs.Post)** attribute) with the following code:
      1. **Note:** This method retrieves through the repository the **Address** entity that corresponds to the **AddressId** received as parameter, and updates it using the form submitted from the browser (**UpdateModel** method). Then it updates the address in the database using the repository, and returns to the customer information view (**Info** view) using the **RedirectToAction** method.
      2. (Code Snippet – *Intro to* *Asp.Net MVC Lab – AddressController Update Action*)
      3. C#
      4. [AcceptVerbs(HttpVerbs.Post)]
      5. **public ActionResult Edit(int addressId, int customerId, FormCollection collection)**
      6. **{**
      7. **try**
      8. **{**
      9. **AddressViewData addressViewData = new AddressViewData();**
      10. **addressViewData.Address = this.repository.GetAddressById(addressId);**
      11. **UpdateModel(addressViewData);**
      12. **this.repository.UpdateAddress();**
      13. **return RedirectToAction("Info", "Customer", new { id = customerId });**
      14. **}**
      15. **catch**
      16. **{**
      17. **return View();**
      18. **}**
      19. **}**
  11. Implement the action method to handle the deletion of a customer address request from the customer information view (**Info** view). To do this, add the following code to the **AddressController** class:
      1. **Note:** This method acts similar as the previous ones, retrieving the **Address** entity and deleting it through the repository, to then return to the customer information view.
      2. (Code Snippet – *Intro to* *Asp.Net MVC Lab* – *AddressController Delete Action*)
      3. C#
      4. **public ActionResult Delete(int addressId, int customerId)**
      5. **{**
      6. **Address address = this.repository.GetAddressById(addressId);**
      7. **this.repository.DeleteAddress(address, customerId);**
      8. **return RedirectToAction("Info", "Customer", new { id = customerId });**
      9. **}**

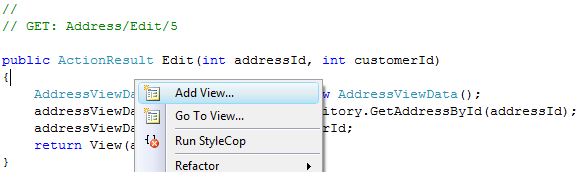
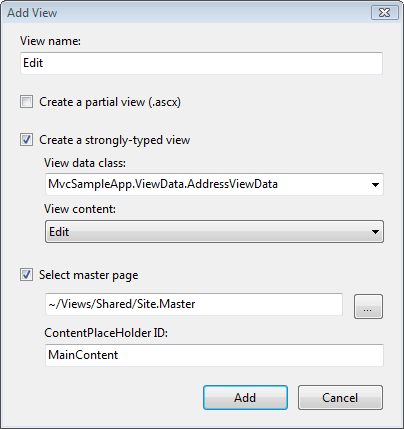
Task 6 – Creating Customer Controller Index View

* 1. In this task you will create one of the views handled by the customer controller. The index view displays a paged list of customers.
  2. Open the **CustomerController** class, right-click inside the **Index** method code, and select **Add View** action from the context menu.
     1. 
     2. Figure
     3. Add View option for Index view
  3. In the **Add View** dialog box specify the following values, and click **Add** to create the view:
     + **View Name**: Index
     + Check the **Create a strongly-typed view** option
       - **View data class**: MvcSampleApp.ViewData.CustomerViewData
       - **View content**: Empty
     + Check the **Select master page** option
       - Leave the default master page value: **~/Views/Shared/Site.Master**
       - **ContentPlaceHolder ID**: MainContent
     1. 
     2. Figure
     3. Adding Index view
  4. Implement the list of customers in the new view. To do this, in **Solution Explorer** double-click **Index.aspx** (under **Views\Customer** folder) file and replace the second *<asp:Content>* tag with the following code:
     1. **Note:** This code loops over the view data collection of customers, printing a link to the customer information view (*Info*) with the customer’s name.
     2. ASP.NET
     3. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     4. **<h2>Customers</h2>**
     5. **<ul>**
     6. **<% foreach (var customer in ViewData.Model.Customers) { %>**
     7. **<li>**
     8. **<%= Html.ActionLink(customer.CompanyName + " - " + customer.FirstName**
     9. **+ " " + customer.LastName, "Info", new { id = customer.CustomerID }) %>**
     10. **</li>**
     11. **<% } %>**
     12. **</ul>**
     13. </asp:Content>
     14. **Note:** The links are constructed using the **Html.ActionLink** helper method. This method helps to dynamically generate HTML hyperlinks that link back to action methods on Controllers using the URL mapping rules of your application. The first argument represents the inner content of the hyperlink to render (the name of the customer in this case); the second argument is the name of the action we’re linking to, and the third argument is an anonymous object that specifies the parameters to construct the URL.
  5. Implement the next and previous page links. To do this paste the following code below the *<ul>* element inserted in the previous step. The links are generated using the *Html.ActionLink* helper method and the **PreviousPage** and **NextPage** properties of the **CustomerViewData**.
     1. ASP.NET
     2. </ul>
     3. **<%=Html.ActionLink("<< Previous Page", "Index", new { page = ViewData.Model.PreviousPage }) %>&nbsp;&nbsp;&nbsp;&nbsp**
     4. **<%=Html.ActionLink("Next Page >>", "Index", new { page = ViewData.Model.NextPage }) %>**
     5. </asp:Content>
     6. **Note:** In addition to using **Html.ActionLink**, ASP.NET MVC has also the **Url.Action** helper method that generates raw string URLs.

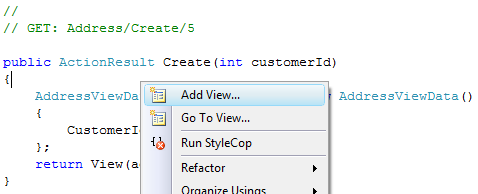
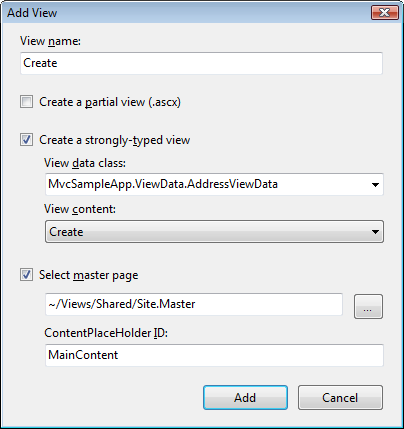
Task 7 – Creating Customer Controller Info View

* 1. In this task you will create other views handled by the customer controller. The **Info** view displays customer’s information, including a list of his/her addresses.
  2. Open the **CustomerController** class, right-click inside the **Info** method code, and select **Add View** action from the context menu.
     1. 
     2. Figure
     3. Add View option for Info view
  3. In the **Add View** dialog box specify the following values:
     + **View Name**: Info
     + Check the Create a strongly-typed view option
       - **View data class**: MvcSampleApp.Models.Customer
       - **View content**: Empty
     + Check the **Select master page** option
       - Leave the default master page value: **~/Views/Shared/Site.Master**
       - **ContentPlaceHolder ID**: MainContent
     1. 
     2. Figure 21
     3. Adding Info view
  4. Add the code to display the customer general information. To do this, in **Solution Explorer** double-click **Info.aspx** (under **Views\Customers** folder), and paste the following code below the second *<asp:Content>* tag to replace the generated code. The customer company’s name, email address, name and phone are displayed in a table using a MVC helper method to display text boxes.
     1. ASP.NET
     2. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     3. **<h2>**
     4. **Customer Information</h2>**
     5. **<fieldset>**
     6. **<p>**
     7. **CompanyName:**
     8. **<%= Html.Encode(Model.CompanyName) %>**
     9. **</p>**
     10. **<p>**
     11. **EmailAddress:**
     12. **<%= Html.Encode(Model.EmailAddress) %>**
     13. **</p>**
     14. **<p>**
     15. **Name:**
     16. **<%= Html.Encode(Model.Title + " " + Model.FirstName + " " + Model.MiddleName + Model.LastName) %>**
     17. **</p>**
     18. **<p>**
     19. **Phone:**
     20. **<%= Html.Encode(Model.Phone) %>**
     21. **</p>**
     22. **</fieldset>**
     23. </asp:Content>
  5. Add the code to display a list of the customer’s addresses. To do this, paste the following code below the *<fieldset>* section inserted in the previous step (and inside the *</asp:Content>* tag).
     1. **Note:** This code loops over the collection of addresses inside the customer entity object printing the addresses and generating links to the **Address** controller edit and delete actions. Finally, a link to add a new address to the customer is inserted.
     2. ASP.NET
     3. </fieldset>
     4. **<h3>**
     5. **Addresses</h3>**
     6. **<ul>**
     7. **<% foreach (var address in Model.CustomerAddress)**
     8. **{ %>**
     9. **<li>**
     10. **<%= address.Address.AddressLine1 + " " + address.Address.AddressLine2 + " - " + address.Address.City %>**
     11. **<%=Html.ActionLink("(Edit)", "Edit", "Address", new { address.AddressID, Model.CustomerID }, null)%>**
     12. **<%=Html.ActionLink("(Delete)", "Delete", "Address", new { address.AddressID, Model.CustomerID }, null)%>**
     13. **</li>**
     14. **<% } %>**
     15. **</ul>**
     16. **<%=Html.ActionLink("Add New Address", "Create", "Address", new { Model.CustomerID }, null)%>**
     17. </asp:Content>

Task 8 – Creating Address Controller Edit View

* 1. In this task you will create the **Edit** view which is handled by the address controller. This view will display information about an existing address, giving the possibility to make changes.
  2. Open the **AddressController** class, right-click inside the **Edit** method code for the **GET** operation, and select **Add View** action from the context menu.
     1. 
     2. Figure
     3. Add View option for Edit view
  3. In the **Add View** dialog box specify the following values, and click **Add**:
     + **View Name**: Edit
     + Check the **Create a strongly-typed view** option
       - **View data class**: MvcSampleApp.ViewData.AddressViewData
       - **View content**: Edit
     + Check the **Select master page** option
       - Leave the default master page value: **~/Views/Shared/Site.Master**
       - **ContentPlaceHolder ID**: MainContent
     1. 
     2. Figure
     3. Adding Edit view
  4. Add the code to display the address information. To do this, in **Solution Explorer** double-click **Edit.aspx** under the **Views\Address** folder to open the file. Paste the following code below the second *<asp:Content>* tag to replace the generated code. The address information is displayed in a table using the **Html.TextBox** helper method to display text boxes, and the **Html.Form** helper method to render a form.
     1. ASP.NET
     2. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     3. **<h2>Editing: <%= Model.Address.AddressLine1 %></h2>**
     4. **<%= Html.ValidationSummary("Edit was unsuccessful. Please correct the errors and try again.") %>**
     5. **<% using (Html.BeginForm()) {%>**
     6. **<fieldset>**
     7. **<legend>Fields</legend>**
     8. **<p>**
     9. **<label for="AddressLine1">Address Line 1:</label>**
     10. **<%=Html.TextBox("Address.AddressLine1")%>**
     11. **</p>**
     12. **<p>**
     13. **<label for="AddressLine2">Address Line 2:</label>**
     14. **<%=Html.TextBox("Address.AddressLine2")%>**
     15. **</p>**
     16. **<p>**
     17. **<label for="City">City:</label>**
     18. **<%=Html.TextBox("Address.City")%>**
     19. **</p>**
     20. **<p>**
     21. **<label for="StateProvince">State/Province:</label>**
     22. **<%=Html.TextBox("Address.StateProvince")%>**
     23. **</p>**
     24. **<p>**
     25. **<label for="PostalCode">Postal Code:</label>**
     26. **<%=Html.TextBox("Address.PostalCode")%>**
     27. **</p>**
     28. **<p>**
     29. **<label for="CountryRegion">Country/Region:</label>**
     30. **<%=Html.TextBox("Address.CountryRegion")%>**
     31. **</p>**
     32. **<p>**
     33. **<input type="submit" value="Save" />**
     34. **</p>**
     35. **</fieldset>**
     36. **<% } %>**
     37. </asp:Content>
     38. **Note:** Notice how the **Html.BeginForm** helper method is used to render a form leveraging the **IDisposable** pattern with the using keyword to auto-terminate the **Form** declaration. The form submitting method call will then be translated into the proper URL form.
     39. **Note:** The **Html.TextBox** helper method used above dynamically generates the HTML code for a text box. It receives as arguments: the html name, the text to display and an anonymous object with desired element’s html attributes.
     40. Notice that in some cases, we’re not specifying the value of the textbox. This is because the **Html.TextBox** helper method will look for a property in ViewData (or ViewData.Model) that has the same name as the name given to the textbox. If it finds one then it will grab that value and use it.

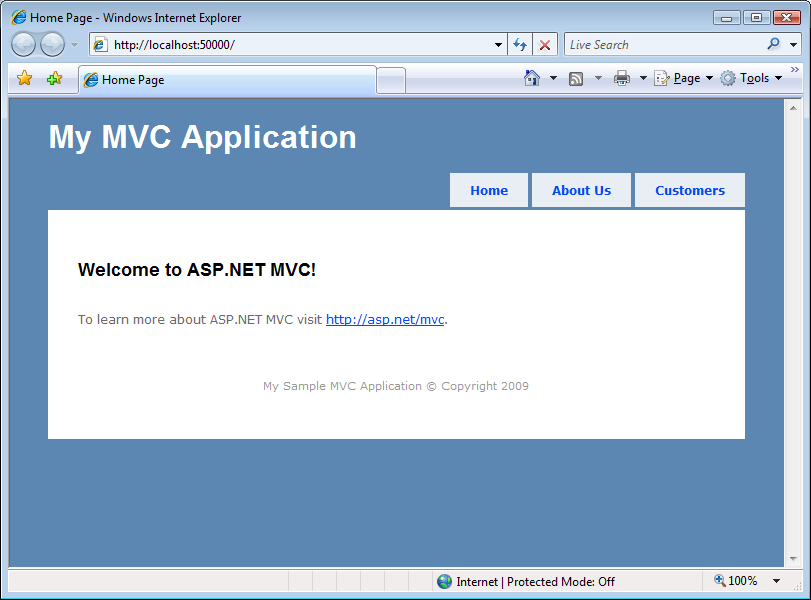
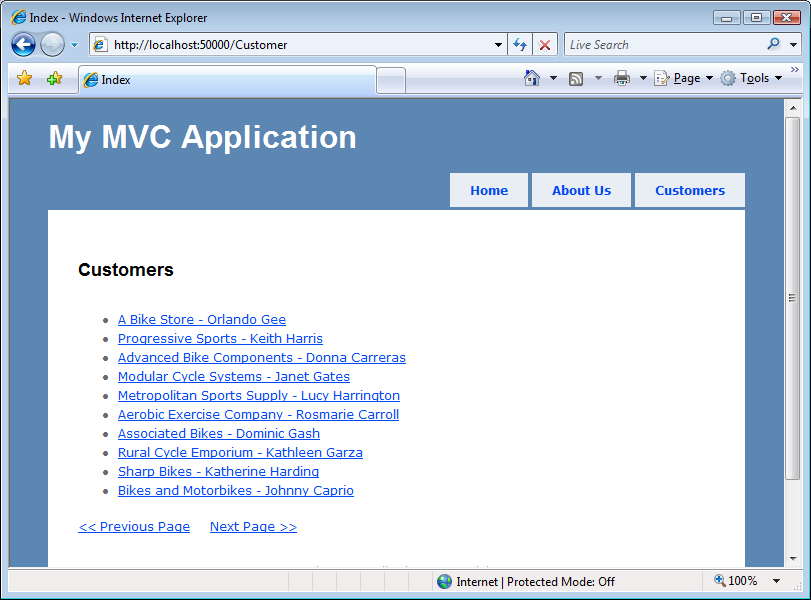
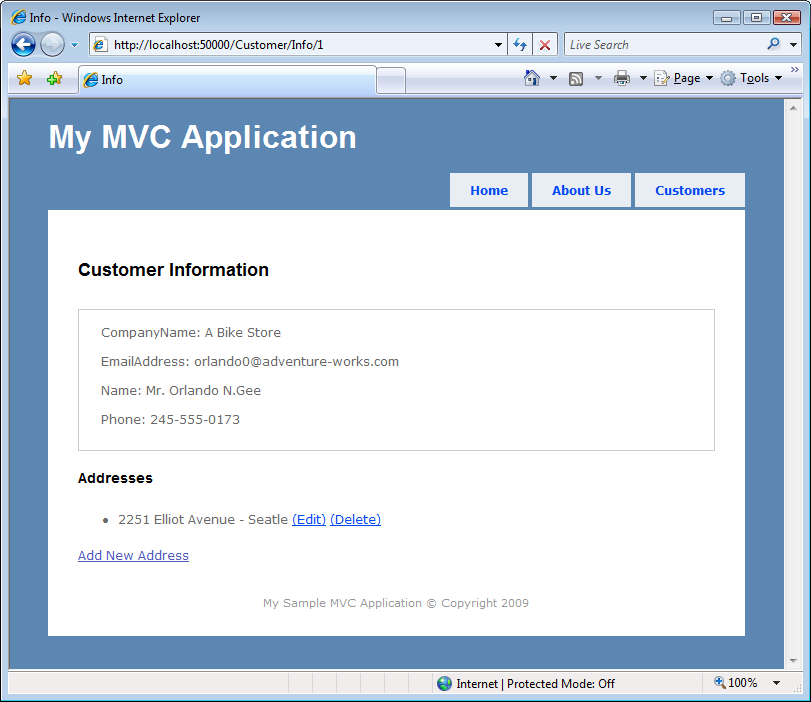
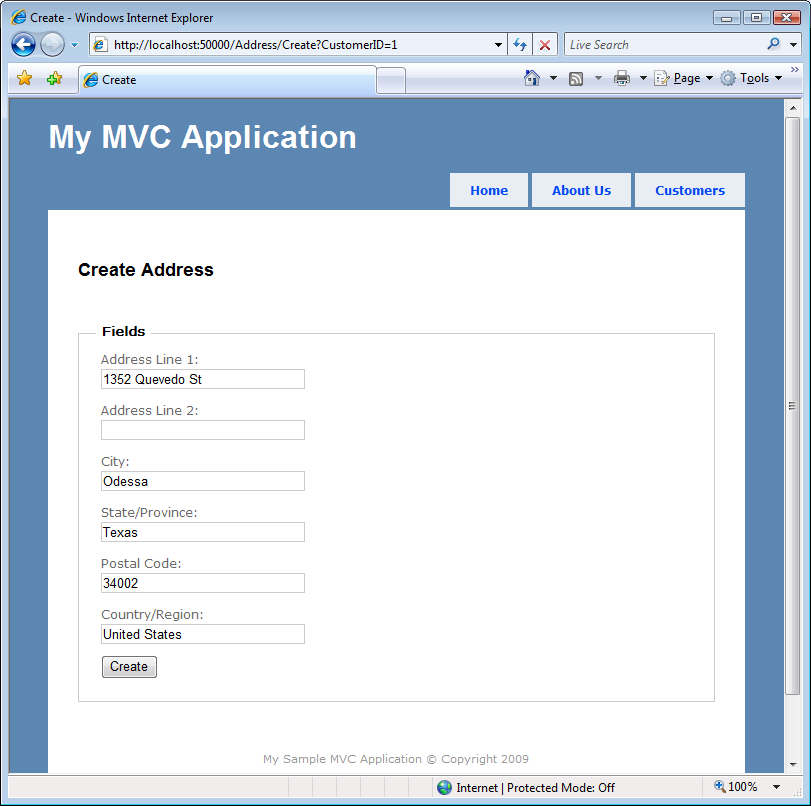
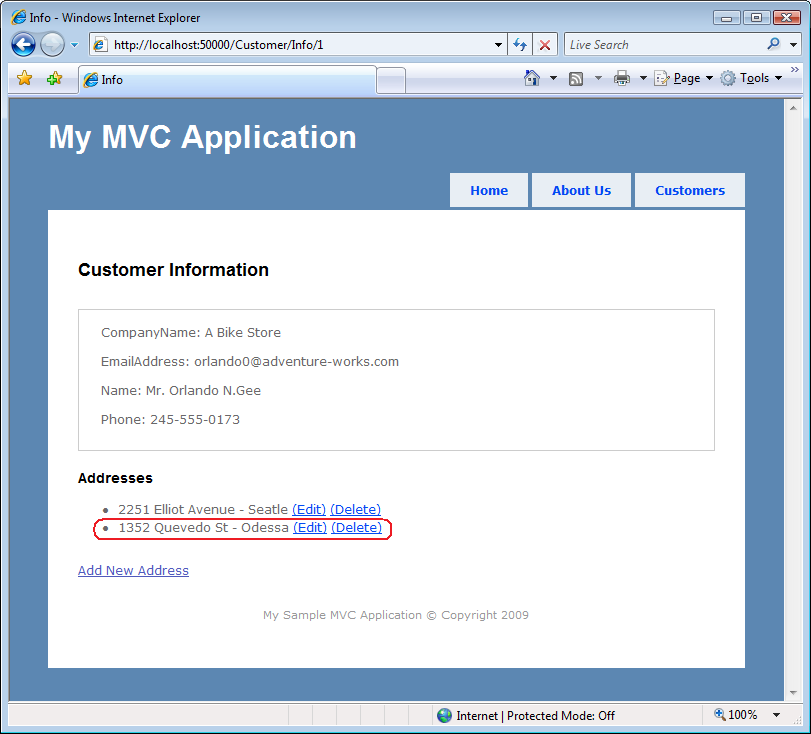
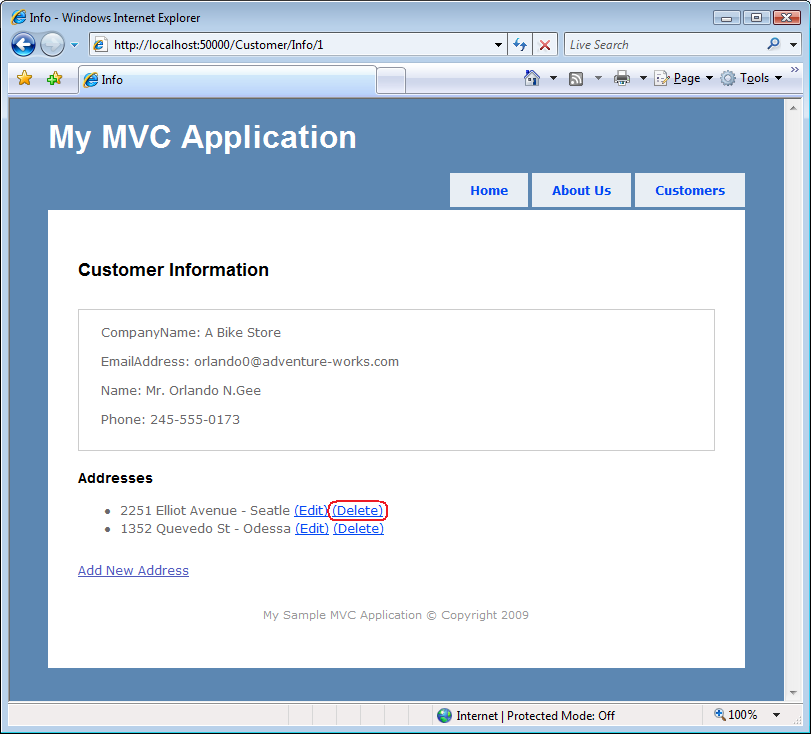
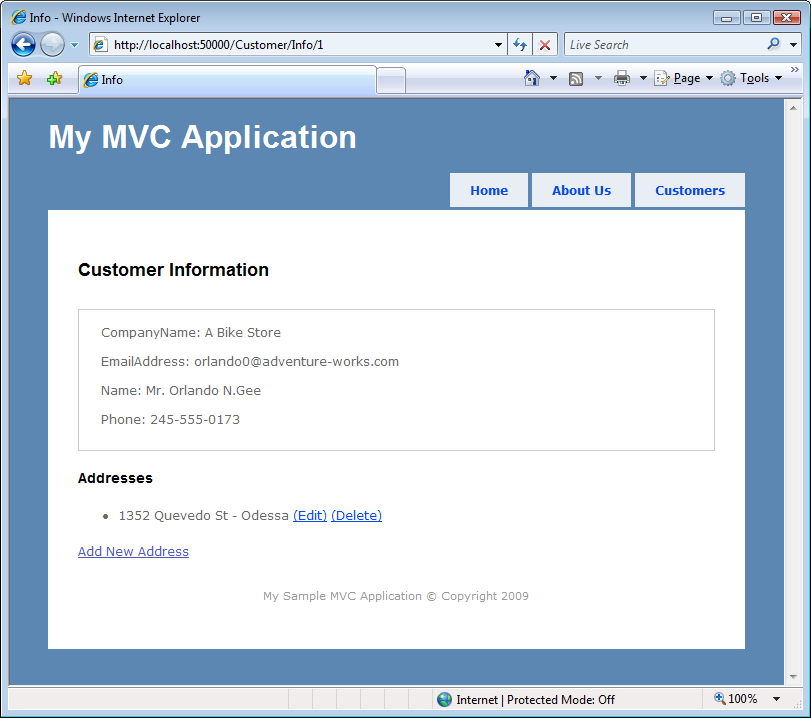
Task 9 – Creating Address Controller New View

* 1. In this task you will create the **Create** view which is handled by the address controller. The **Create** view displays a blank form to create a new address for the selected customer.
  2. Open the **AddressController** class, right-click inside the **Create** method code for the **GET** operation, and select **Add View** action from the context menu.
     1. 
     2. Figure 24
     3. Add View option for Create view
  3. In the **Add View** dialog box specify the following values, and click **Add** to create the view:
     + **View Name**: Create
     + Check the **Create a strongly-typed view** option
       - **View data class**: MvcSampleApp.ViewData.AddressViewData
       - **View content**: Create
     + Check the **Select master page** option
       - Leave the default master page value: **~/Views/Shared/Site.Master**
       - **ContentPlaceHolder ID**: MainContent
     1. 
     2. Figure 25
     3. Adding Create view
  4. Implement the blank form in the view to capture user input for the new address. To do this, in **Solution Explorer** double-click **Create.aspx** (under the **Views\Address** folder) to open the file. Replace the second *<asp:Content>* tag with the following code:
     1. ASP.NET
     2. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     3. **<h2>**
     4. **Create Address**
     5. **</h2>**
     6. **<%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>**
     7. **<% using (Html.BeginForm())**
     8. **{%>**
     9. **<fieldset>**
     10. **<legend>Fields</legend>**
     11. **<p>**
     12. **<label for="AddressLine1">**
     13. **Address Line 1:</label>**
     14. **<%= Html.TextBox("Address.AddressLine1") %>**
     15. **</p>**
     16. **<p>**
     17. **<label for="AddressLine2">**
     18. **Address Line 2:</label>**
     19. **<%= Html.TextBox("Address.AddressLine2") %>**
     20. **</p>**
     21. **<p>**
     22. **<label for="City">**
     23. **City:</label>**
     24. **<%= Html.TextBox("Address.City")%>**
     25. **</p>**
     26. **<p>**
     27. **<label for="StateProvince">**
     28. **State/Province:</label>**
     29. **<%= Html.TextBox("Address.StateProvince")%>**
     30. **</p>**
     31. **<p>**
     32. **<label for="PostalCode">**
     33. **Postal Code:</label>**
     34. **<%= Html.TextBox("Address.PostalCode")%>**
     35. **</p>**
     36. **<p>**
     37. **<label for="CountryRegion">**
     38. **Country/Region:</label>**
     39. **<%= Html.TextBox("Address.CountryRegion")%>**
     40. **</p>**
     41. **<p>**
     42. **<input type="submit" value="Create" />**
     43. **</p>**
     44. **</fieldset>**
     45. **<% } %>**
     46. </asp:Content>

## Exercise 2: Verification

* 1. In order to verify that you have correctly performed all steps of exercise one, proceed as follows:

### Verification 1

* 1. In this verification you will navigate the MCV Web Application, testing **Home** and **Customer** controllers.
  2. To test the MVC Web Application you need to start a new instance of the **MvcSampleApp** project. To do this, in **Solution Explorer** right-click **MvcSampleApp** project, point to **Debug** and select **Start New Instance**.
     1. **Note:** If the dialog **Debugging Not Enabled** appears, select **Modify the Web.config file to enable debugging** and click **OK**.
     2. You will be directed to[**http://localhost:50000/**](http://localhost:50000/). This is the home page of the MVC Web Application, and it is generated by the ASP.NET MVC framework by calling **HomeController** controller’s **Index** methodthatrenders the **Index** view.
     3. **Note:** In this case **Index** method only renders the view, since no data is needed by the Home controller’s **Index** View.
     4. 
     5. Figure
     6. MVC Sample Application home page
  3. To browse customers, click **Customers** link on page header. You will be redirected to the following address in the web browser <http://localhost:50000/Customer> which invokes **Index** method on the **CustomerController**. The following output should appear. Previous and Next Page links are provided to navigate through all customers. The page showed by default is page one.
     1. 
     2. Figure
     3. Customers Index page
  4. Click on a name to view a specific customer detail. For example click the first customer. You will be redirected to <http://localhost:50000/Customer/Info/1> (where “1” is the **Customer Id**).
     1. **Note:** The method **Info** on **CustomerController** is invoked, which retrieves customer data, and pass it to the **Info View** when rendering it. Details on the given customer and its currently associated addresses will be shown.
     2. Remember the default route created in **Global.asax** was “{controller}/{action}/{id}”. So in this case “1” is the id and “Info” is the action.
     3. 
     4. Figure
     5. Customer Info page
     6. **Note:** In the addresses section of customer’s information, next to each address there are two links, one for editing the address and the other to delete and there is also an **Add New Address** link to create and associate a new address to the current customer.
  5. Click on the **Add New Address** link to create a new address for the current customer.
     1. **Note:** Clicking on Add New Address triggers the **Create** method on **AddressController** which renders the **Create** view.
     2. When the save button is pressed on the view, the form submits to the **Create** method. This method receives the address information and performs the insertion before redirecting to customer information page.
  6. Fill the form, and click **Create**. You will be redirected to the customer information page where the new address will be shown.
     1. 
     2. Figure
     3. Creating a new Address
     4. 
     5. Figure
     6. Viewing the recently inserted Address
  7. Delete an address. To do this, click the **Delete** link on any of the addresses of the list.
     1. 
     2. Figure
     3. Deleting an address
     4. **Note:** Clicking on **Delete** link will trigger **Delete** method on **AddressController**, which doesn’t render any view; instead, it performs deletion of selected address and then reloads customer information page to show the changes.
     5. 
     6. Figure
     7. Address deleted
  8. You can also try the *Edit* operation on this application since it is similar to *Create*.
  9. Close the application to end this verification.

Exercise 3: Testing an ASP.NET MVC Application

* 1. Separation of application tasks (input logic, business logic, and UI logic) in MVC ASP.NET applications enables testability, and test-driven development (TDD) by default. All core contracts within the MVC framework are interface-based and can be tested by using mock objects, which are simulated objects that imitate the behavior of actual objects in the application. You can unit-test the application without having to run the controllers in an ASP.NET process, which makes unit testing fast and flexible. You can use any unit-testing framework that is compatible with the .NET Framework.
  2. Even though this exercise does not follow strictly the Test-Driven Development approach you will learn what is needed and how to perform unit testing while creating an ASP.NET MVC application.

Task 1 – Opening the ASP.NET MVC Application

* 1. In this task you will obtain the necessary files to be able to do the exercise and open the Visual Studio begin solution.
  2. To able to test in isolation ASP.NET MVC routes, this exercise uses a mock framework for the .NET platform called **Moq**, which is the same mock framework used by the ASP.NET MVC team.
  3. Download the **Moq** Library. To do this, follow this link: <http://code.google.com/p/moq/> and download the **Moq** Library 3.0 version binaries.
  4. Once the download is finished, extract and copy the **Moq.dll** file into **AspNetMvc\Assets\** folder**.**
  5. Open Microsoft Visual Studio 2008 from **Start | All Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008**. Open the solution file **MvcSampleApp.sln** located under **AspNetMvc\Ex03-TestingMvcApp\begin\**.

Task 2 – Testing Application Routes

* 1. In this task you will learn how to write unit tests for the route matching rules defined in **Global.asax**.
  2. Add a reference to the **Moq** Library core assembly. To do this, in **Solution Explorer**, right-click **MvcSampleApp.Test** project and select **Add Reference**. In the **Add Reference** dialog box, select the **Browse** tab, browse to **AspNetMvc\Assets** folder select **Moq.dll**. Click **OK** to add the reference.
  3. Import the route tests outline. To do this, right-click on the **MvcSampleApp.Test** project, and select **Add Existing Item**. Navigate to the **AspNetMvc\Assets** folder and select the **RouteTests.cs** file. Double-click on the **RouteTests.cs** file to open it up.
  4. Add the following namespace directive to **RouteTests.cs**.
     1. C#
     2. **using Moq;**
  5. Implement the **GetRouteDataForUrl** method. To do this, paste the following code (shown in **bold**) inside the method:
     1. **Note:** This method emulates the mechanism that performs the ASP.NET Routing engine when it receives a URL and should return the matching route data from the loaded application route collection. Given a **RouteData** collection and a URL, it uses a mocked **HttpContextBase** object and the URL parameter to find the matching **RouteData** in the **RouteCollection**.
     2. (Code Snippet – *Intro to* *Asp.Net MVC Lab* – *GetRouteDataForUrl*)
     3. C#
     4. private static RouteData GetRouteDataForUrl(RouteCollection routes, string url)
     5. {
     6. **var mockRequest = new Mock<HttpRequestBase>();**
     7. **mockRequest.Setup(r => r.AppRelativeCurrentExecutionFilePath).Returns(url);**
     8. **mockRequest.Setup(r => r.PathInfo).Returns(String.Empty);**
     9. **var mockContext = new Mock<HttpContextBase>();**
     10. **mockContext.Setup(c => c.Request).Returns(mockRequest.Object);**
     11. **RouteData routeData = routes.GetRouteData(mockContext.Object);**
     12. **return routeData;**
     13. }
     14. **Note:** For more information about how the **GetRouteDataForUrl** method works. Please check the Appendix section.
  6. Start implementing a test method for the customer default route (*“~/Customer”*) that maps to the **Customer** controller **Index** view. Paste the following code inside **ShouldCustomerTakeDefaultRoute** test method:
     1. **Note:** This section creates a new route collection, and fills it with all the routes used in the previous exercise by calling the **MvcApplication.RegisterRoutes()**.
     2. C#
     3. [TestMethod]
     4. public void ShouldCustomerTakeDefaultRoute()
     5. {
     6. **RouteCollection routes = new RouteCollection();**
     7. **MvcApplication.RegisterRoutes(routes);**
     8. }
  7. Add code to call the **GetRouteDataForUrl** method and return the matching route for the “*~/Customer”* Url. Paste the following code below the previous one.
     1. C#
     2. [TestMethod]
     3. public void ShouldCustomerTakeDefaultRoute()
     4. {
     5. RouteCollection routes = new RouteCollection();
     6. MvcApplication.RegisterRoutes(routes);
     8. **RouteData routeData = GetRouteDataForUrl(routes, "~/Customer");**
     9. }
  8. Add the “asserts” to the method. Paste the following code below the previous one. The code asserts that:
     1. A route has been found.
     2. The controller that will be called is the **Customer** controller.
     3. The action method that will be called is **Index**.
     4. (Code Snippet – Intro to Asp.Net MVC Lab – ShouldCustomerTakeDefaultRoute)
     5. C#
     6. [TestMethod]
     7. public void ShouldCustomerTakeDefaultRoute()
     8. {
     9. RouteCollection routes = new RouteCollection();
     10. MvcApplication.RegisterRoutes(routes);
     12. RouteData routeData = GetRouteDataForUrl(routes, "~/Customers");
     14. **Assert.IsNotNull(routeData, "Should have found the route");**
     15. **Assert.AreEqual("Customer", routeData.Values["controller"], "Customer controller expected");**
     16. **Assert.AreEqual("Index", routeData.Values["action"], "Index action expected");**
     17. }
  9. Implement a test method for theroute that maps to **Customer** controller **Info** action (e.g. “*~/Customer/Info/1*”). To do this, paste the following code into **ShouldCustomerTakeInfoRoute** test method. This method does the same as **ShouldCustomerTakeDefaultRoute** and asserts that:
     1. A route has been found.
     2. The controller that is called is the **Customer** controller.
     3. The **Id** parameter value is one.
     4. The action method that is called is **Info**.
     5. (Code Snippet – *Intro to* *Asp.Net MVC Lab* – *ShouldCustomerTakeInfoRoute*)
     6. C#
     7. [TestMethod]
     8. public void ShouldCustomerTakeInfoRoute()
     9. {
     10. **RouteCollection routes = new RouteCollection();**
     11. **MvcApplication.RegisterRoutes(routes);**
     12. **RouteData routeData = GetRouteDataForUrl(routes, "~/Customer/Info/1");**
     13. **Assert.IsNotNull(routeData, "Should have found the route");**
     14. **Assert.AreEqual("Customer", routeData.Values["controller"], "Customer controller expected");**
     15. **Assert.AreEqual("1", routeData.Values["id"], "Customer ID = 1 expected");**
     16. **Assert.AreEqual("Info", routeData.Values["action"], "Info action expected");**
     17. }
  10. Implement a test method for theroute that maps to **Address** controller **Create** view (e.g. “~/Address/Create/1”). Paste the following code into **ShouldAddressTakeCreateRoute** test method. This method does the same as the previous one and asserts that:
      1. A route has been found.
      2. The controller that is called is the **Address** controller.
      3. The **Id** parameter value is one.
      4. The action method that is called is **Create**.
      5. (Code Snippet – *Intro to* *Asp.Net MVC Lab* – *ShouldAddressTakeNewRoute*)
      6. C#
      7. [TestMethod]
      8. public void ShouldAddressTakeCreateRoute()
      9. {
      10. **RouteCollection routes = new RouteCollection();**
      11. **MvcApplication.RegisterRoutes(routes);**
      12. **RouteData routeData = GetRouteDataForUrl(routes, "~/Address/Create/1");**
      13. **Assert.IsNotNull(routeData, "Should have found the route");**
      14. **Assert.AreEqual("Address", routeData.Values["Controller"], "Address controller expected");**
      15. **Assert.AreEqual("1", routeData.Values["Id"], "Customer ID = 1 expected");**
      16. **Assert.AreEqual("Create", routeData.Values["action"], "Create action expected");**
      17. }
      18. **Note:** If you want to test these methods run successfully you can now follow the steps in Verification 1.

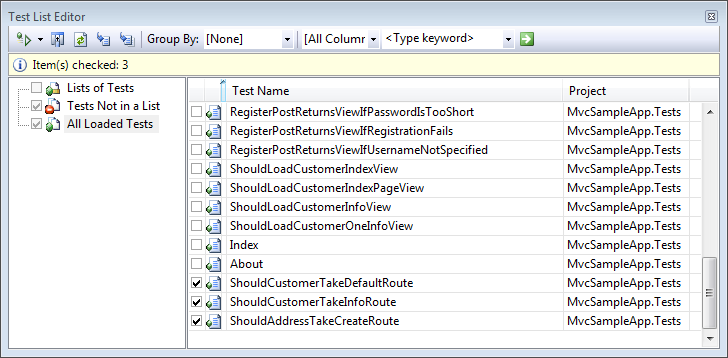
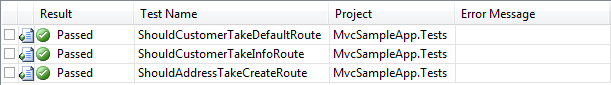
Task 3 – Testing Customer Controller

* 1. In this task you will create unit tests for the **Customer** controller. In the application, the **Customer** controller uses **AdventureWorks** entity data model to interact with the database. The idea in this task is to also mock entity data model so as to have full control over the tests. If any of these tests fail, you will be sure that the defect was produced within the **Controller** component rather than in the **AdventureWorks** entity data model.
  2. Import the **CustomerController** test class. To do this, right-click the **Controllers** folder under the **MvcSampleApp.Test** project and select **Add Existing Item**. Navigate to the **AspNetMvc\Assets** directory and select the **CustomerControllerTest.cs** file.
  3. Import the stub version for the **AdventureWorks** repository. To do this, right-click the **MvcSampleApp.Test** project and select **Add Existing Item**. Navigate to the **AspNetMvc\Assets** directory and select the **AdventureWorksRepositoryStub.cs** file.
     1. **Note:** The **AdventureWorksRepositoryStub** class implements the **IAdventureWorksRepository** interface. You will include this interface in a further step.
  4. Add a reference to the **System.Data.Entity** assembly. To do this, in **Solution Explorer**, right-click **MvcSampleApp.Test** project and select **Add Reference**. In the **Add Reference** dialog box, select the **.NET** tab, select the **System.Data.Entity** component. Click **OK** to add the reference.
  5. Import the **AdventureWorks** repository interface. To do this, right-click the **Models** folder under **MvcSampleApp** project and select Add Existing Item. Navigate to the **AspNetMvc\Assets** directory and select the **IAdventureWorksRepository.cs** file.
  6. Modify the **AdventureWorksRepository** class to implement the AdventureWorks repository interface. To do this, open the **AdventureWorksRepository.cs** file under the **Models** folder in the **MvcSampleApp** project and replace the class declaration using the following code.
     1. C#
     2. public class AdventureWorksRepository **: IAdventureWorksRepository**
  7. Modify the **CustomerController** class to implement two constructor methods. The first method will create an instance of **AdventureWorksRepository** and the second one will accept an **IAdventureWorksRepository** implementation. To do this, open the **CustomerController.cs** file under the **Controllers** folder in the **MvcSampleApp** project and replace the repository field declaration with the following constructor methods.
     1. C#
     2. public class CustomerController : Controller
     3. {
     4. **private IAdventureWorksRepository repository;**
     5. **public CustomerController()**
     6. **{**
     7. **this.repository = new AdventureWorksRepository();**
     8. **}**
     9. **public CustomerController(IAdventureWorksRepository repository)**
     10. **{**
     11. **this.repository = repository;**
     12. **}**
     13. …
     14. }
  8. Double-click **CustomerControllerTest.cs** for edit. Implement a test method to verify that the **Customer** controller **Index** action method renders the **Index** view, passing a view data that is not null. Paste the following code into **ShouldLoadCustomerIndexView** test method in the **CustomerControllerTests** class.
     1. (Code Snippet – *Intro to* *Asp.Net MVC Lab* – *ShouldLoadCustomerIndexView*)
     2. C#
     3. [TestMethod]
     4. public void ShouldLoadCustomerIndexView()
     5. {
     6. **CustomerController controller = new CustomerController(new AdventureWorksRepositoryStub());**
     7. **ViewResult result = controller.Index(1) as ViewResult;**
     8. **Assert.IsTrue(string.IsNullOrEmpty(result.ViewName));**
     9. **Assert.IsNotNull(result.ViewData);**
     10. }
     11. **Note:** The method uses a repository stub version (**AdventureWorksRepositoryStub**) that implements only the **GetCustomers** method (returning two empty customers) and **GetCustomersById** method. Check the **AdventureWorksRepositoryStub.cs** file under the test project for implementation details.
     12. Notice that you’re checking for an *Empty* or *Null* string value from the rendered view name. This is because when the controller calls the **View** method without specifying a view name, the resulting **ViewResult**’s **ViewName** property will be null, indicating that no explicit view was specified, and that the view with the same name as the controller action method should be rendered. You can explicitly specify the **ViewName** by calling an overloaded version of **View()**.
  9. Implement a test method that verifies that the **CustomerViewData** is properly filled in the **Index** method of the **Customers** controller. Paste the following code into **ShouldLoadCustomerIndexPageView** test method.
     1. (Code Snippet – *Intro to* *Asp.Net MVC Lab* – *ShouldLoadCustomerIndexPageView*)
     2. C#
     3. [TestMethod]
     4. public void ShouldLoadCustomerIndexPageView()
     5. {
     6. **CustomerController controller = new CustomerController(new AdventureWorksRepositoryStub());**
     7. **ViewResult result = controller.Index(1) as ViewResult;**
     9. **Assert.IsInstanceOfType(result.ViewData.Model, typeof(CustomerViewData));**
     10. **CustomerViewData customerViewData = result.ViewData.Model as CustomerViewData;**
     11. **Assert.AreEqual(2, customerViewData.NextPage, "Page 2 expected");**
     12. **Assert.AreEqual(2, customerViewData.Customers.Count(), "2 Customers expected");**
     13. }
  10. Implement a test method that verifies that the **Customer** controller **Info** action method renders the **Info** view, passing a view data which is not null. Paste the following code into **ShouldLoadCustomerInfoView** test method.
      1. (Code Snippet – *Intro to* *Asp.Net MVC Lab* – *ShouldLoadCustomerInfoView*)
      2. C#
      3. [TestMethod]
      4. public void ShouldLoadCustomerInfoView()
      5. {
      6. **CustomerController controller = new CustomerController(new AdventureWorksRepositoryStub());**
      7. **ViewResult result = controller.Info(1) as ViewResult;**
      8. **Assert.IsTrue(string.IsNullOrEmpty(result.ViewName));**
      9. **Assert.IsNotNull(result.ViewData);**
      10. }
  11. Implement a test method that verifies that the **Customer** entity view data is properly filled in the **Info** method of the **Customers** controller. Paste the following code into **ShouldLoadCustomerOneInfoView** test method.
      1. (Code Snippet – *Intro to* *Asp.Net MVC Lab* – *ShouldLoadCustomerOneInfoView*)
      2. C#
      3. [TestMethod]
      4. public void ShouldLoadCustomerOneInfoView()
      5. {
      6. **CustomerController controller = new CustomerController(new AdventureWorksRepositoryStub());**
      7. **ViewResult result = controller.Info(1) as ViewResult;**
      8. **Assert.IsInstanceOfType(result.ViewData.Model, typeof(Customer));**
      9. **Customer customer = result.ViewData.Model as Customer;**
      10. **Assert.AreEqual(1, customer.CustomerID, "CustomerId 1 expected");**
      11. }

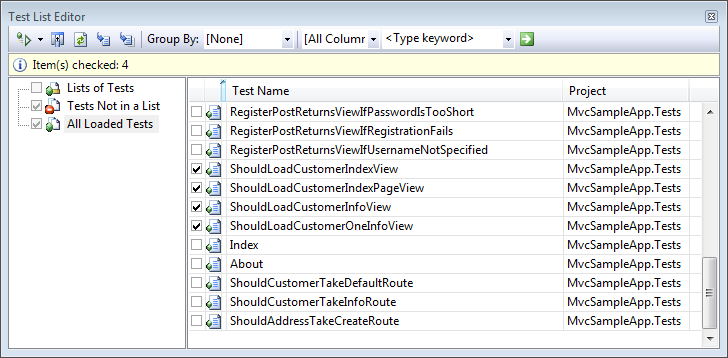
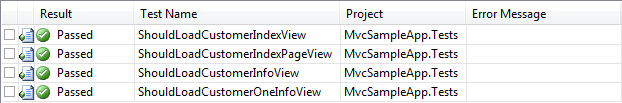
## Exercise 3: Verification

* 1. In order to verify that you have correctly performed all steps of exercise two, proceed as follows:

### Verification 1

* 1. In this verification you will make sure all the route tests implemented pass successfully.
  2. Open the solution **MvcSampleApp.sln** in Visual Studio if not already opened.
  3. Press **Ctrl + Shift + B** to build the solution.
  4. In **Test** menu, point to **Windows** and select **Test List Editor**.
  5. Select the following tests:
     1. ShouldCustomerTakeDefaultRoute
     2. ShouldCustomerTakeInfoRoute
     3. ShouldAddressTakeCreateRoute
     4. 
     5. Figure
     6. Selecting tests to run
     7. **Note:** The ASP.NET MVC Test Project Template adds 25 pre-built unit tests that verify the behavior of the **AccountsController** class (which is a controller added to the project by default to handle login and account management scenarios.) Review the **AccountControllerTest.cs** file for detailed examples of controller unit testing.
  6. In **Test** menu, point to **Run** and select **Tests in Current Context**.
  7. Make sure that all tests pass.
     1. 
     2. Figure
     3. Tests pass

### Verification 2

* 1. In this verification you will make sure that all the tests implemented for the **Customer** controller pass successfully.
  2. Open the solution **MvcSampleApp.sln** in Visual Studio if not already opened.
  3. Press **Ctrl + Shift + B** to build the solution.
  4. In **Test** menu, point to **Windows** and select **Test List Editor**.
  5. Select the following tests:
     1. ShouldLoadCustomerIndexView
     2. ShouldLoadCustomerIndexPageView
     3. ShouldLoadCustomerInfoView
     4. ShouldLoadCustomerOneInfoView
     5. 
     6. Figure
     7. Selecting tests to run
  6. In **Test** menu, point to **Run** and select **Tests in Current Context**.
  7. Make sure that all tests pass.
     1. 
     2. Figure

Tests pass

Summary

* 1. By completing this Hands-On Lab you have learnt which are the core elements of an MVC application, and how they interact. You have also learnt how to create and test an ASP.NET MVC Application.

Appendix

Moq Library

* 1. It is a library that allows you, among other things, to create mock objects and set their expected behavior in an easy way.
  2. The **GetRouteDataForUrl** method used in this exercise works in the following way.
  3. C#
  4. private static RouteData GetRouteDataForUrl(RouteCollection routes, string url)
  5. {
  6. var mockRequest = new Mock<HttpRequestBase>();
  7. mockRequest.Setup(r => r.AppRelativeCurrentExecutionFilePath).Returns(url);
  8. mockRequest.Setup(r => r.PathInfo).Returns(String.Empty);
  9. var mockContext = new Mock<HttpContextBase>();
  10. mockContext.Setup(c => c.Request).Returns(mockRequest.Object);
  11. RouteData routeData = routes.GetRouteData(mockContext.Object);
  12. return routeData;
  13. }
  14. The first thing is to create your mock object, which is done using the **Mock<T>** class. You need to create a mock request object that will allow faking an HTTP request.
      1. C#
      2. var mockRequest = new Mock<HttpRequestBase>();
  15. Now that you have the mock request, you will set up some of the behavior you need out of the request. This includes making sure that the fake request appears to be asking for the URL passed into the method.
      1. C#
      2. mockRequest.Setup(r => r.AppRelativeCurrentExecutionFilePath).Returns(url);

mockRequest.Setup(r => r.PathInfo).Returns(String.Empty);

* 1. Now that you have the mock request configured, you need to create a mock HTTP context and assign the mock request to it.
     1. C#
     2. var mockContext = new Mock<HttpContextBase>();
     3. mockContext.Setup(c => c.Request).Returns(mockRequest.Object);
  2. Finally, with the mock context, you can request the route data from the passed in **RouteCollection** using the mock context.
     1. C#
     2. RouteData routeData = routes.GetRouteData(mockContext.Object);
     3. return routeData;