Hands-On Lab

Introduction to ASP.NET MVC

Lab version: 1.0.0

Last updated: 12/10/2010

developer & platform evangelism
## Contents

**OVERVIEW**........................................................................................................................................... 3

**EXERCISE 1: CREATING AN ASP.NET MVC APPLICATION**................................................................. 5
  Task 1 – Creating an ASP.NET MVC Web Application Project ....................................................... 5
  Task 2 – Exploring the Solution Structure ..................................................................................... 9
  Task 3 – Understanding the Controllers ....................................................................................... 12
  Task 4 – Understanding the Views .................................................................................................. 16
  Task 5 – Understanding the ASP.NET URL Routing ............................................................... 17

**EXERCISE 2: DEVELOPING AN ASP.NET MVC APPLICATION**..................................................... 20
  Task 1 – Creating an Entity Data Model ....................................................................................... 21
  Task 2 – Implementing Customer View Data ............................................................................... 25
  Task 3 – Implementing Address View Data .................................................................................. 27
  Task 4 – Implementing Customer Controller ............................................................................. 29
  Task 5 – Implementing Address Controller .............................................................................. 32
  Task 6 – Creating Customer Controller Index View ............................................................... 37
  Task 7 – Creating Customer Controller Info View ..................................................................... 42
  Task 8 – Creating Address Controller Edit View ...................................................................... 46
  Task 9 – Creating Address Controller New View .................................................................... 50

  Exercise 2: Verification ........................................................................................................... 55

**EXERCISE 3: TESTING AN ASP.NET MVC APPLICATION**........................................................... 62
  Task 1 – Opening the ASP.NET MVC Application ..................................................................... 63
  Task 2 – Testing Application Routes ........................................................................................... 63
  Task 3 – Testing Customer Controller ....................................................................................... 69

  Exercise 3: Verification ........................................................................................................... 74

**SUMMARY**........................................................................................................................................... 77

**APPENDIX**........................................................................................................................................ 77
  Moq Library .................................................................................................................................. 77
Overview

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.

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.

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.

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

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
You must have the following items to complete this lab:

- Microsoft Visual Studio 2010
- ASP.NET MVC 2
- Microsoft SQL Server 2005 or Microsoft SQL Server 2008 (Express edition or above)
- Adventure Works sample database

**Note:** the Dependency Checker will copy the Adventure Works sample database file on each exercise solution folder. If you wish you can copy it manually from the Assets folder.

Setup
All the requisites for this lab are verified using the Dependency Checker. To make sure that everything is correctly configured, follow these steps:

**Note:** To perform the setup steps you need to run the scripts in a command window with administrator privileges.

1. Run the dependency checker if you have not done so previously. To do this, browse to \Source\Setup folder of this lab and run the CheckDependencies.cmd script. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.

**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. If you cannot find the snippets when you write the solutions, make sure you installed the code snippets inside Visual Studio 2010 Code Snippets Repository.

Exercises
This Hands-On Lab is comprised by the following exercises:

1. Creating an ASP.NET MVC Application
2. Developing an ASP.NET MVC Application
3. Testing an ASP.NET MVC Application
Estimated time to complete this lab: **90 minutes**.

**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.

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.

**Note:** Each exercise contains a Visual Basic and a C# version; Inside the End/Begin solution folder you will find two folders: VB, containing the Visual Basic version of the exercise, and C#, containing the C# version of it.

---

**Next Step**

**Exercise 1: Creating an ASP.NET MVC Application**

**Exercise 1: Creating an ASP.NET MVC Application**

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.

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**

In this task you will create and configure an empty ASP.NET MVC application project using the MVC Visual Studio template.

1. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
2. On the File menu, point to New, and click Project.

3. In the New Project dialog box make sure that .NET Framework 4 is selected and select either Visual C# or Visual Basic then select the ASP.NET MVC 2 Web Application project type. You may set the location to the Source\Ex01-CreatingMvcApp\begin (choosing the folder that matches the language of your preference) which is the provided folder for this lab.

4. Change the Name to MvcSampleApp and click OK.

Figure 1
Create New Project Dialog Box (C#)
5. 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.

**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.
Create Unit Test Project

Would you like to create a unit test project for this application?

- Yes, create a unit test project

Test project name:

MvcSampleApp.Test

Test framework:

Visual Studio Unit Test

- No, do not create a unit test project

Figure 3
Create Unit Tests Dialog Box

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.

5. Configure the web site to use port 50000. This step is needed for consistency with the end solution provided.

a. To do this, in Solution Explorer, right-click MvcSampleApp project and in the context menu select Properties.

b. In the Property pages open the Web tab.

c. In the Servers section select Specific Port.

d. Set the port number to 50000.

e. Press Ctrl+S to save changes.
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.
Figure 5

ASP.NET MVC Solution Structure (C#)
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 an 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.
**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.

**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 3 – Understanding the Controllers**

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.

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.
1. Open the **HomeController**. To do this, in the Solution Explorer, double-click the **HomeController.cs (C#)** or **HomeController.cs (Visual Basic)** file under **Controllers** folder. You will see the following code:

```
[HandleError]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewData["Message"] = "Welcome to ASP.NET MVC!";

        return View();
    }

    public ActionResult About()
    {
        return View();
    }
}
```

**Figure 7**

*Viewing the Controller’s auto generated code (C#)*
<HandleError/>

Public Class HomeController

    Function Index() As ActionResult
        ViewData(”Message”) = ”Welcome to ASP.NET MVC!”
        Return View()
    End Function

    Function About() As ActionResult
        Return View()
    End Function
End Class

Figure 8
Viewing the Controller’s auto generated code (C#)

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.

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.

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.

ASP.NET MVC framework supports several types of action results including:
- ViewResult: Represents HTML and markup.
- EmptyResult: Represents no result.
- RedirectResult: Represents a redirection to a new URL.
- JsonResult: Represents a JavaScript Object Notation result that can be used in an AJAX application.
- JavaScriptResult: Represents a JavaScript script.
- ContentResult: Represents a text result.
- FileContentResult: Represents a downloadable file (with the binary content).
- FilePathResult: Represents a downloadable file (with a path).
- FileStreamResult: Represents a downloadable file (with a file stream).
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:

- **View**: Returns a **ViewResult** action result.
- **Redirect**: Returns a **RedirectResult** action result.
- **RedirectToAction**: Returns a **RedirectToRouteResult** action result (redirects to the specified action).
- **RedirectToRoute**: Returns a **RedirectToRouteResult** action result (redirects to the specified route).
- **Json**: Returns a **JsonResult** action result.
- **JavaScriptResult**: Returns a **JavaScriptResult**.
- **Content**: Returns a **ContentResult** action result.
- **File**: Returns a **FileContentResult**, **FilePathResult**, or **FileStreamResult** depending on the parameters passed to the method.

For more information, see [http://www.asp.net/learn/mvc/tutorial-03-cs.aspx](http://www.asp.net/learn/mvc/tutorial-03-cs.aspx) for a C# version or [http://www.asp.net/learn/mvc/tutorial-03-vb.aspx](http://www.asp.net/learn/mvc/tutorial-03-vb.aspx) for the Visual Basic version.

2. Views should only render their output using the view-specific data provided by the Controller class. In the ASP.NET MVC Framework this is called view-specific data, **ViewData**.

    **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.

```csharp
    public ActionResult Index()
    {
        ViewData["Message"] = "Welcome to ASP.NET MVC!";

        return View();
    }
```

**Figure 9**

*Using the ViewData dictionary (C#)*
Task 4 – Understanding the Views

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.

   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.

   ```csharp
   Function Index() As ActionResult
       ViewData("Message") = "Welcome to ASP.NET MVC!"
       Return View()
   End Function
   
   Figure 10
   Using the ViewData dictionary (Visual Basic)
   
   Figure 11
   Viewing the View's auto generated code (C#)
   ```
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.

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>** (C#) or **ViewPage(Of T)** (Visual Basic), 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 5 – Understanding the ASP.NET URL Routing**

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.

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.

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).
1. 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.

   **Note:** If the dialog **Debugging Not Enabled** appears, select **Modify the Web.config file to enable debugging** and click OK.

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.

```csharp
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.sax/{pathInfo}");

    routes.MapRoute(
        "Default", // Route name
        "{controller}/{action}/{id}", // URL with parameters
        new { controller = "Home", action = "Index", id = "" } // Parameter defaults
    );
}
```

**Figure 13**

*Viewing the default registered route in Global.asax (C#)*

```vbnet
Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
    routes.IgnoreRoute("{resource}.sax/{pathInfo}")

    ' MapRoute takes the following parameters, in order:
    ' (1) Route name
    ' (2) URL with parameters
    ' (3) Parameter defaults
    routes.MapRoute( _
        "Default", _
        "{controller}/{action}/{id}", _
        New With {.controller = "Home", .action = "Index", .id = ""} _
    )

End Sub
```

**Figure 14**

*Viewing the default registered route in Global.asax (Visual Basic)*
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.

![My MVC Application](image)

**Figure 15**  
*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.

2. 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](http://localhost:50000/Home/About) which invokes **About** method on the **Home Controller**.
Exercise 2: Developing an ASP.NET MVC Application

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.

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 addresses. 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.
Task 1 – Creating an Entity Data Model

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 the solution file MvcSampleApp.sln located under AspNetMvc\Source\Ex02-DevelopingMvcApp\begin\ (choosing the folder that matches the language of your preference.) Alternatively, you may continue working with the solution obtained after completing the previous exercise.

3. 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.

4. Import the provided Site.Master. To do this, right-click the Shared folder, point to Add / Existing Item. In the Add Existing Item dialog, browse to Source\Assets folder (choosing the folder that matches the language of your preference) and select Site.Master in the Shared folder. Click Add.

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.

5. 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.

6. In the Add New Item dialog box select ADO.NET Entity Data Model. Specify a Name value of AdventureWorks.edmx, and then click Add.
Figure 17
Adding the ADO.NET Entity Data Model (C#)

Figure 18
Adding the ADO.NET Entity Data Model (Visual Basic)
7. After the **Entity Data Model Wizard** opens, select **Generate From Database** and click **Next**.

8. Specify the Database connection. To do this, follow these steps:
   a. Click **New Connection**
   b. In **Choose Data Source** dialog, select **Microsoft SQL Server Database File (SqlClient)** as **Data Source** and click **Continue**.
   c. Click the **Browse** button to set the database file name to use.
   d. Browse to the **Source\Assets** folder and select the **AdventureWorksLT.mdf** file.
   e. Click **OK** to save the connection.

![Connection Properties](image)

**Figure 19**
*Specify the database connection*

9. Back on the Entity Data Model Wizard click **Next**.
10. You will be prompted if you want to copy the file inside your project directory and change the connection string accordingly. Click Yes to proceed.

11. Include only the following tables from all the proposed Database objects:

   - Address (SalesLT)
   - Customer (SalesLT)
   - CustomerAddress (SalesLT)

12. Unselect the Pluralize or singularize generated object names option.

13. Leave the Model Namespace by default and click Finish.

![Entity Data Model Wizard]

**Figure 20**

*Choose the Database Objects to Include in the Model*

14. Add the AdventureWorksRepository. The repository exposes methods to retrieve entities from the model generating a level of abstraction from the underlying data model. To do this, right-click the Models folder of MvcSampleApp project in Solution Explorer, point to Add / Existing
Item. In the Add Existing Item dialog, browse to Source\Assets folder (choosing the folder that matches the language of your preference) and select AdventureWorksRepository.cs and click Add.

Note: For information see ADO.NET Entity Framework.

Task 2 – Implementing Customer View Data

In this task you will create the View Data that will be used in the customer controller to display the paged list of customers.

1. Create the ViewData folder. To do this, right-click MvcSampleApp project, point to Add and select New Folder. Set ViewData as the folder name.

2. Create the CustomerViewData 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 CustomerViewData.cs (C#) or CustomerViewData.vb (Visual Basic), and then click Add.

Figure 21
Adding CustomerViewData class (C#)
4. **Only for C# users:** In `CustomerViewData.cs`, replace all the using statements created by default with the following code.

```csharp
using System;
using System.Web;
using System.Web.Mvc;
using System.Collections.Generic;
using MvcSampleApp.Models;
```

5. 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.

(Code Snippet – *Intro to Asp.Net MVC Lab - CustomerViewData Properties CSharp*)

```csharp
public class CustomerViewData
{
    public IEnumerable<Customer> Customers
    {
        get;
        set;
```
Visual Basic

Public Class CustomerViewData
    Public Property Customers As IEnumerable(Of Customer)
    Public Property PreviousPage As Integer
    Public Property NextPage As Integer
End Class

Task 3 – Implementing Address View Data

In this task you will create the view data that will be used in the address controller to allow the editing of an address.

1. Create the AddressViewData class. Right-click the ViewData folder, point to Add and select New Item.

2. In the Add New Item dialog box select Class. Specify a Name value of AddressViewData.cs (C#) or AddressViewData.vb (Visual Basic), and then click Add.
Figure 23
Adding AddressViewData class (C#)

Figure 24
Adding AddressViewData class (Visual Basic)
3. **Only for C# users:** 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.

```csharp
using System;
using System.Web;
using System.Web.Mvc;
using MvcSampleApp.Models;
```

4. 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.

(Code Snippet - Intro to Asp.Net MVC Lab – AddressViewData Properties CSharp)

```csharp
public class AddressViewData
{
    public Address Address
    {
        get;
        set;
    }

    public int CustomerId
    {
        get;
        set;
    }
}
```

(Code Snippet - Intro to Asp.Net MVC Lab – AddressViewData Properties VB)

```vbnet
Public Class AddressViewData
    Public Property Address As Address
    Public Property CustomerId As Integer
End Class
```

**Task 4 – Implementing Customer Controller**
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).

1. Create an empty MVC controller class. In Solution Explorer right-click the Controllers folder, point to Add and select Controller.

2. Specify a Controller Name value of CustomerController, and then click Add.

![Add Controller](image)

**Figure 25**
Adding CustomerController controller

3. **Only for C# users**: In CustomerController.cs, add the following namespace directives:

```csharp
using MvcSampleApp.Models;
using MvcSampleApp.ViewData;
```

4. Instantiate a repository to act as a data access service. To do this, paste the following code inside the CustomerController class.

```csharp
private AdventureWorksRepository repository = new AdventureWorksRepository();
```

```vbnet
Private repository As New AdventureWorksRepository()
```

5. Implement the action method to handle the CustomerController Index view. To do this, replace the default Index method with the following code.

```
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.

(Code Snippet – Intro to Asp.Net MVC Lab - CustomerController Index Action CSharp)

C#

```csharp
public ActionResult Index(int? page)
{
    var viewData = new CustomerViewData();
    int currentPage = page ?? 0;
    viewData.Customers = this.repository.GetCustomers(currentPage, 10);
    viewData.NextPage = currentPage + 1;
    viewData.PreviousPage = (currentPage <= 0) ? 0 : currentPage - 1;
    return View(viewData);
}
```

(Code Snippet – Intro to Asp.Net MVC Lab - CustomerController Index Action VB)

Visual Basic

```vb
Public Function Index(ByVal page As Nullable(Of Integer)) As ActionResult
    Dim viewData = New CustomerViewData()
    Dim currentPage As Integer = If(page, 0)
    viewData.Customers = Me.repository.GetCustomers(currentPage, 10)
    viewData.NextPage = currentPage + 1
    viewData.PreviousPage = If((currentPage <= 0), 0, currentPage - 1)
    Return View(viewData)
End Function
```

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.

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). The MVC framework will either pass in a value if a page number is present in the URL - or pass in null if not.

6. 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.
Task 5 – Implementing Address Controller

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 responsible of handling the form submissions sent back from these two views.

1. Create an empty MVC controller class. In Solution Explorer right-click the Controllers folder, point to Add and select Controller.

2. 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.

3. Only for C# users: 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.

```csharp
using MvcSampleApp.Models;
using MvcSampleApp.ViewData;
```

4. Instantiate a repository to act as a data access service. To do this, paste the following code inside the AddressController class.

```csharp
public class AddressController : Controller
{
    private AdventureWorksRepository repository = new AdventureWorksRepository();
    ...
}
```

5. Remove the Index and Details methods created by default.

6. 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:

   **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.

   (Code Snippet – Intro to Asp.Net MVC Lab – AddressController New Action CSharp)

```csharp
public ActionResult Create(int customerId)
{
    AddressViewData addressViewData = new AddressViewData()
    {
        CustomerId = customerId
    }
```
Visual Basic

Function Create(ByVal customerId As Integer) As ActionResult
    Dim addressViewData = New AddressViewData With
        {.CustomerId = customerId}
    Return View(addressViewData)
End Function

7. 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:

    Note: This method receives the customerId 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.

(CODE SNIPPET -- Intro to Asp.Net MVC Lab -- AddressController Create Action CSharp)

C#

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(int customerId, FormCollection collection)
{
    try
    {
        AddressViewData addressViewData = new AddressViewData();
        UpdateModel(addressViewData);
        this.repository.AddAddress(addressViewData.Address, customerId);
        return RedirectToAction("Info", "Customer", new { id = customerId });
    }
    catch
    {
        return View();
    }
}
Visual Basic

```vbnet
<AcceptVerbs(HttpVerbs.Post)> _
Function Create(ByVal customerId As Integer, ByVal collection As FormCollection) As ActionResult
    Try
        Dim addressViewData = New AddressViewData()
        UpdateModel(addressViewData)
        Me.repository.AddAddress(addressViewData.Address, customerId)
        Return RedirectToAction("Info", "Customer", New With {.id = customerId})
    Catch
        Return View()
    End Try
End Function
```

**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.

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.

8. 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:

```csharp
public ActionResult Edit(int addressId, int customerId)
{
    AddressViewData addressViewData = new AddressViewData();
    addressViewData.Address = this.repository.GetAddressById(addressId);
    addressViewData.CustomerId = customerId;
    return View(addressViewData);
}
```

**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.
Visual Basic

Public Function Edit(ByVal addressId As Integer, ByVal customerId As Integer) As ActionResult
    Dim addressViewData As New AddressViewData()
    addressViewData.Address = Me.repository.GetAddressById(addressId)
    addressViewData.CustomerId = customerId
    Return View(addressViewData)
End Function

9. 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:

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.

C#

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int addressId, int customerId, FormCollection collection)
{
    try
    {
        AddressViewData addressViewData = new AddressViewData();
        addressViewData.Address = this.repository.GetAddressById(addressId);
        UpdateModel(addressViewData);
        this.repository.UpdateAddress();
        return RedirectToAction("Info", "Customer", new { id = customerId });
    }
    catch
    {
        return View();
    }
}
Visual Basic

<AcceptVerbs(HttpVerbs.Post)>  
Public Function Edit(ByVal addressId As Integer, ByVal customerId As Integer, ByVal collection As FormCollection) As ActionResult  
    Try  
        Dim addressViewData As New AddressViewData()  
        addressViewData.Address = Me.repository.GetAddressById(addressId)  
        UpdateModel(addressViewData)  
        Me.repository.UpdateAddress()  
        Return RedirectToAction("Info", "Customer", New With {.id = customerId})  
    Catch  
        Return View()  
    End Try  
End Function

10. 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:

    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.

    (Code Snippet – Intro to Asp.Net MVC Lab – AddressController Delete Action C#)

C#

public ActionResult Delete(int addressId, int customerId)  
{  
    Address address = this.repository.GetAddressById(addressId);  
    this.repository.DeleteAddress(address, customerId);  
    return RedirectToAction("Info", "Customer", new { id = customerId });  
}

(Code Snippet – Intro to Asp.Net MVC Lab – AddressController Delete Action VB)

C#

Public Function Delete(ByVal addressId As Integer, ByVal customerId As Integer) As ActionResult  
    Dim address = Me.repository.GetAddressById(addressId)  
    Me.repository.DeleteAddress(address, customerId)  
    Return RedirectToAction("Info", "Customer", New With {.id = customerId})  
End Function

Task 6 – Creating Customer Controller Index View
In this task you will create one of the views handled by the customer controller. The index view displays a paged list of customers.

1. Open the `CustomerController` class, right-click inside the `Index` method code, and select `Add View` action from the context menu.

```csharp
public ActionResult Index(int? page)
{
    var viewData = new CustomerViewData();
    int currentPage = page ?? 0;
    viewData.Customers = this.repository.Get();
    viewData.NextPage = currentPage + 1;
    viewData.PreviousPage = (currentPage < 1) ? null : currentPage - 1;
    return View(viewData);
}
```

**Figure 27**
Add View option for Index view (C#)

```vbnet
Public Function Index(ByVal page As Nullable(Of Integer)) As ActionResult
Dim viewData As New CustomerViewData()
Dim currentPage As Integer = If(page, 1, viewData.Customers = Me.repository.Get())
viewData.NextPage = currentPage + 1
viewData.PreviousPage = If((currentPage > 1) ? null : currentPage - 1)
Return View(viewData)
End Function
```

**Figure 28**
Add View option for Index view (Visual Basic)

2. 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 (C#) or MvcSampleApp.CustomerViewData (Visual Basic)
     - **View content**: Empty
   - Check the `Select master page` option
     - Leave the default master page value: `~/Views/Shared/Site.Master`
   - **ContentPlaceHolder ID**: MainContent
Figure 29

Adding Index view (C#)
3. 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:

**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.

### ASP.NET (C#)

```csharp
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Customers</h2>
    <ul>
        <% foreach (var customer in ViewData.Model.Customers) { %>
            <li>
                <%= Html.ActionLink(customer.CompanyName + " - " + customer.FirstName + " " + customer.LastName, "Info", new { id = customer.CustomerID }) %>
            </li>
        <% } %>
    </ul>
</asp:Content>
```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>customers</h2>
    <ul>
      <%For Each customer In ViewData.Model.Customers%>
      <li><%= Html.ActionLink(customer.CompanyName + " - " + customer.FirstName + " " + customer.LastName, "Info", New With {.id = customer.CustomerID})%></li>
      <%Next%>
    </ul>
</asp:Content>

**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 you’re linking to, and the third argument is an anonymous object that specifies the parameters to construct the URL.

4. 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`.

```csharp
<%=Html.ActionLink("<< Previous Page", "Index", new { page = ViewData.Model.PreviousPage }) %>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<%=Html.ActionLink("Next Page >>", "Index", new { page = ViewData.Model.NextPage }) %>
</asp:Content>
```

```vbnet
<%= Html.ActionLink("<< Previous Page", "Index", New With {.page = ViewData.Model.PreviousPage})%>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
```
Task 7 – Creating Customer Controller Info View

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.

1. Open the **CustomerController** class, right-click inside the **Info** method code, and select **Add View** action from the context menu.

   ```csharp
   public ActionResult Info(int id)
   {
       var customer = customerRepository.GetCustomerById(id);
       return View(customer);
   }
   ```

   **Figure 31**
   Add View option for Info view (C#)

   ```vbnet
   Public Function Info(ByVal id As Integer) As ActionResult
       Dim customer = Me.CustomerRepository.GetCustomerById(id)
       Return View(customer)
   End Function
   ```

   **Figure 32**
   Add View option for Info view (Visual Basic)

2. 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 (C#) or MvcSampleApp.Customer (Visual Basic)
     - **View content**: Empty
   - Check the **Select master page** option
     - Leave the default master page value: `~/Views/Shared/Site.Master`
Figure 33

Adding Info view (C#)
3. 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.

**ASP.NET (both C# and Visual Basic)**

```<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <h2>Customer Information</h2>
  <fieldset>
    <p>CompanyName: <%= Html.Encode(Model.CompanyName) %></p>
    <p>EmailAddress: <%= Html.Encode(Model.EmailAddress) %></p>
  </fieldset>
</asp:Content>```
4. 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).

   **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.

   **ASP.NET (C#)**

```
</fieldset>
<h3>Addresses</h3>
<ul>
  <% foreach (var address in Model.CustomerAddress) %>
    <li>
      <%= address.Address.AddressLine1 + " " + address.Address.AddressLine2 + " - " + address.Address.City %>
    </li>
  <% } %>
</ul>
</asp:Content>
```

   **ASP.NET (Visual Basic)**

```
</fieldset>
<% For Each address In Model.CustomerAddress%>
<li>
```

Task 8 – Creating Address Controller Edit View

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.

1. Open the **AddressController** class, right-click inside the **Edit** method code for the **GET** operation, and select **Add View** action from the context menu.

   ![Add View option for Edit view (C#)]

   **Figure 35**
   Add View option for Edit view (C#)

   ![Add View option for Edit view (Visual Basic)]

   **Figure 36**
   Add View option for Edit view (Visual Basic)

2. 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 (C#) or MvcSampleApp.AddressViewData (Visual Basic)
- **View content**: Edit
  - Check the **Select master page** option
  - Leave the default master page value: `~/Views/Shared/Site.Master`
- **ContentPlaceHolder ID**: MainContent

*Figure 37*
*Adding Edit view (C#)*
3. 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.

**ASP.NET (C#)**

    <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
        <h2>Editing: <%= Model.Address.AddressLine1 %></h2>
        <%= Html.ValidationSummary("Edit was unsuccessful. Please correct the errors and try again.") %>
        <% using (Html.BeginForm()) {%>
            <fieldset>
                <legend>Fields</legend>
                <p>
                    <label for="AddressLine1">Address Line 1:</label>
                </p>
            </fieldset>
        <%} %>
    </asp:Content>
<% using(Html.BeginForm()) %>
<% undisponible %>
<legend>Fields</legend>
<p>
    <label for="AddressLine1">Address Line 1:</label>
    <%=Html.TextBox("Address.AddressLine1")%>
</p>
<p>
    <label for="AddressLine2">Address Line 2:</label>
    <%=Html.TextBox("Address.AddressLine2")%>
</p>
<p>
    <label for="City">City:</label>
    <%=Html.TextBox("Address.City")%>
</p>
<p>
    <label for="StateProvince">State/Province:</label>
    <%=Html.TextBox("Address.StateProvince")%>
</p>
<p>
    <label for="PostalCode">Postal Code:</label>
    <%=Html.TextBox("Address.PostalCode")%>
</p>
<p>
    <label for="CountryRegion">Country/Region:</label>
    <%=Html.TextBox("Address.CountryRegion")%>
</p>
<p>
    <input type="submit" value="Save" />
</p>
</fieldset>

ASP.NET (Visual Basic)
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Editing: <%= Model.Address.AddressLine1 %></h2>

    <%= Html.ValidationSummary("Edit was unsuccessful. Please correct the errors and try again.") %>

    <% using(Html.BeginForm()) %>
    <legend>Fields</legend>
    <p>
        <label for="AddressLine1">Address Line 1:</label>
        <%=Html.TextBox("Address.AddressLine1")%>
    </p>
    <p>
        <label for="AddressLine2">Address Line 2:</label>
        <%=Html.TextBox("Address.AddressLine2")%>
    </p>
    <p>
        <label for="City">City:</label>
        <%=Html.TextBox("Address.City")%>
    </p>
    <p>
        <label for="StateProvince">State/Province:</label>
        <%=Html.TextBox("Address.StateProvince")%>
    </p>
    <p>
        <label for="PostalCode">Postal Code:</label>
        <%=Html.TextBox("Address.PostalCode")%>
    </p>
    <p>
        <label for="CountryRegion">Country/Region:</label>
        <%=Html.TextBox("Address.CountryRegion")%>
    </p>
    <p>
        <input type="submit" value="Save" />
    </p>
</asp:Content>
Task 9 – Creating Address Controller New View

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.

1. Open the AddressController class, right-click inside the Create method code for the GET operation, and select Add View action from the context menu.
2. 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` (C#) or `MvcSampleApp.AddressViewData` (Visual Basic)
     - **View content**: Create
   - Check the **Select master page** option
     - Leave the default master page value: `~/Views/Shared/Site.Master`
     - **ContentPlaceHolder ID**: MainContent
Figure 41

Adding Create view (C#)
3. 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:

**ASP.NET (C#)**

```csharp
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <h2>Create Address</h2>
  <%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>
  <% using (Html.BeginForm()) { %>
    <fieldset>
      <legend>Fields</legend>
      <p>
        <label for="AddressLine1">Address Line 1:</label>
        <%= Html.TextBox("Address.AddressLine1") %>
      </p>
    </fieldset>
  <% } %>
</asp:Content>
```
<p>
<p>
<label for="AddressLine2">
Address Line 2:
</label>

<%= Html.TextBox("Address.AddressLine2") %>
</p>

</p>

<p>
<label for="City">
City:
</label>

<%= Html.TextBox("Address.City") %>
</p>

</p>

<p>
<label for="StateProvince">
State/Province:
</label>

<%= Html.TextBox("Address.StateProvince") %>
</p>

</p>

<p>
<label for="PostalCode">
Postal Code:
</label>

<%= Html.TextBox("Address.PostalCode") %>
</p>

</p>

<p>
<label for="CountryRegion">
Country/Region:
</label>

<%= Html.TextBox("Address.CountryRegion") %>
</p>

</p>

<input type="submit" value="Create" />

</fieldset>

</asp:Content>

ASP.NET (Visual Basic)

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
<h2>Create Address</h2>

<%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>

<% Using Html.BeginForm()%>
<fieldset>
<legend>Fields</legend>
<p>
<label for="AddressLine1">
Address Line 1:
</label>

<%= Html.TextBox("Address.AddressLine1") %>
</p>

</fieldset>
</asp:Content>
Next Step

Exercise 2: Verification

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

Verification

In this verification you will navigate the MCV Web Application, testing Home and Customer controllers.
1. 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.

Note: If the dialog Debugging Not Enabled appears, select Modify the Web.config file to enable debugging and click OK.

You will be directed to 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** method that renders the **Index** view.

**Note:** In this case **Index** method only renders the view, since no data is needed by the Home controller’s **Index** View.

2. 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.

![My MVC Application](image)

**Figure 44**

*Customers Index page*

3. 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](http://localhost:50000/Customer/Info/1) (where “1” is the **Customer Id**).

**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.

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.
4. Click on the Add New Address link to create a new address for the current customer.

Note: Clicking on Add New Address triggers the Create method on AddressController which renders the Create view.

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.
5. Fill the form, and click **Create**. You will be redirected to the customer information page where the new address will be shown.

*Figure 46*

*Creating a new Address*
Delete an address. To do this, click the **Delete** link on any of the addresses of the list.
Deleting an address

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.
7. You can also try the *Edit* operation on this application since it is similar to *Create*.
8. Close the application to end this verification.

**Next Step**

Exercise 3: Testing an ASP.NET MVC Application

**Exercise 3: Testing an ASP.NET MVC Application**
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.

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**

In this task you will obtain the necessary files to be able to do the exercise and open the Visual Studio begin solution.

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.

1. Download the **Moq** Library. To do this, follow this link: [http://code.google.com/p/moq/](http://code.google.com/p/moq/) and download the **Moq** Library 3.0 version binaries.
2. Once the download is finished, extract and replace the **Moq.dll** file into **AspNetMvc\Source\Assets\** folder.
3. Open Microsoft Visual Studio 2010 from **Start | All Programs | Microsoft Visual Studio 2010 | Microsoft Visual Studio 2010**.
4. Open the solution file **MvcSampleApp.sln** located under **AspNetMvc\Source\Ex03-TestingMvcApp\begin\** (choosing the folder that matches the language of your preference.) Alternatively, you may continue working with the solution obtained after completing the previous exercise.

**Task 2 – Testing Application Routes**

In this task you will learn how to write unit tests for the route matching rules defined in **Global.asax**.

1. 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\Source\Assets** folder select **Moq.dll**. Click **OK** to add the reference.
2. Import the route tests outline. To do this, right-click on the **MvcSampleApp.Test** project, point to **Add / Existing Item**. In the **Add Existing Item** dialog, browse to **Source\Assets** folder (choosing the folder that matches the language of your preference) and select **RouteTests.cs (C#) or RouteTests.vb (Visual Basic.)** Click **Add**. Double-click on the recently added file to open it up.
3. Add the following namespace directive to **RouteTests.cs (C#) or RouteTests.vb (Visual Basic).**
using Moq;

**Visual Basic**

Imports Moq

4. Implement the **GetRouteDataForUrl** method. To do this, paste the following code (shown in **bold**) inside the method:

**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**.

(Code Snippet – Intro to Asp.Net MVC Lab – GetRouteDataForUrl CSharp)

**C#**

```csharp
private static RouteData GetRouteDataForUrl(RouteCollection routes, string url)
{
    var mockRequest = new Mock<HttpRequestBase>();
    mockRequest.Setup(r => r.AppRelativeCurrentExecutionFilePath).Returns(url);
    mockRequest.Setup(r => r.PathInfo).Returns(String.Empty);

    var mockContext = new Mock<HttpContextBase>();
    mockContext.Setup(c => c.Request).Returns(mockRequest.Object);

    RouteData routeData = routes.GetRouteData(mockContext.Object);
    return routeData;
}
```

(Code Snippet – Intro to Asp.Net MVC Lab – GetRouteDataForUrl VB)

**Visual Basic**

Private Shared Function GetRouteDataForUrl(ByVal routes As RouteCollection, ByVal url As String) As RouteData
    Dim mockRequest = New Mock(Of HttpRequestBase)()
    mockRequest.Setup(Function(r) r.AppRelativeCurrentExecutionFilePath).Returns(url)
    mockRequest.Setup(Function(r) r.PathInfo).Returns(String.Empty)

    Dim mockContext = New Mock(Of HttpContextBase)()
    mockContext.Setup(Function(c) c.Request).Returns(mockRequest.Object)
Dim routeData As RouteData = routes.GetRouteData(mockContext.Object)
Return routeData
End Function

**Note:** For more information about how the `GetRouteDataForUrl` method works. Please check the [Appendix](#) section.

5. Start implementing a test method for the customer default route ("~/Customer") that maps to the Customer controller Index view. Paste the following code inside the `ShouldCustomerTakeDefaultRoute` test method:

   **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()`.

   **C#**
   ```csharp
   [TestMethod]
   public void ShouldCustomerTakeDefaultRoute()
   {
       RouteCollection routes = new RouteCollection();
       MvcApplication.RegisterRoutes(routes);
   }
   ``

   **Visual Basic**
   ```vb
   <TestMethod()> _
   Public Sub ShouldCustomerTakeDefaultRoute()
   Dim routes As New RouteCollection()
   MvcApplication.RegisterRoutes(routes)
   End Sub
   ```

6. Add code to call the `GetRouteDataForUrl` method and return the matching route for the "~/Customer" Url. Paste the following code below the previous one.

   **C#**
   ```csharp
   [TestMethod]
   public void ShouldCustomerTakeDefaultRoute()
   {
       RouteCollection routes = new RouteCollection();
       MvcApplication.RegisterRoutes(routes);

       RouteData routeData = GetRouteDataForUrl(routes, "~/Customer");
   }```
7. Add the “asserts” to the method. Paste the following code below the previous one. The code asserts that:

   a. A route has been found.
   b. The controller that will be called is the Customer controller.
   c. The action method that will be called is Index.

**Visual Basic**

```vb
<TestMethod()> _
Public Sub ShouldCustomerTakeDefaultRoute()
    Dim routes As New RouteCollection()
    MvcApplication.RegisterRoutes(routes)

    Dim routeData = GetRouteDataForUrl(routes, "~/Customer")
End Sub
```

**C#**

```csharp
[TestMethod]
public void ShouldCustomerTakeDefaultRoute()
{
    RouteCollection routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);

    RouteData routeData = GetRouteDataForUrl(routes, "~/Customers");

    Assert.IsNotNull(routeData, "Should have found the route");
    Assert.AreEqual("Customer", routeData.Values["controller"], "Customer controller expected");
    Assert.AreEqual("Index", routeData.Values["action"], "Index action expected");
}
```

**Visual Basic**

```vb
<TestMethod()> _
Public Sub ShouldCustomerTakeDefaultRoute()
    Dim routes As New RouteCollection()
    MvcApplication.RegisterRoutes(routes)

    Dim routeData = GetRouteDataForUrl(routes, "~/Customer")
End Sub
```
Dim routeData = GetRouteDataForUrl(routes, "~/Customer")

Assert.IsNotNull(routeData, "Should have found the route")
Assert.AreEqual("Customer", routeData.Values("controller"), "Customer controller expected")
Assert.AreEqual("Index", routeData.Values("action"), "Index action expected")
End Sub

8. Implement a test method for the route 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:

- A route has been found.
- The controller that is called is the Customer controller.
- The Id parameter value is one.
- The action method that is called is Info.

(Code Snippet – Intro to Asp.Net MVC Lab – ShouldCustomerTakeInfoRoute CSharp)

C#

[TestMethod]
public void ShouldCustomerTakeInfoRoute()
{
    RouteCollection routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);

    RouteData routeData = GetRouteDataForUrl(routes, "~/Customer/Info/1");

    Assert.IsNotNull(routeData, "Should have found the route");
    Assert.AreEqual("Customer", routeData.Values["controller"], "Customer controller expected");
    Assert.AreEqual("1", routeData.Values["id"], "Customer ID = 1 expected");
    Assert.AreEqual("Info", routeData.Values["action"], "Info action expected");
}

(Code Snippet – Intro to Asp.Net MVC Lab – ShouldCustomerTakeInfoRoute VB)

Visual Basic

<TestMethod()> _
Public Sub ShouldCustomerTakeInfoRoute()
    Dim routes As New RouteCollection()
    MvcApplication.RegisterRoutes(routes)

    Dim routeData = GetRouteDataForUrl(routes, "~/Customer/Info/1")

    Assert.IsNotNull(routeData, "Should have found the route");
    Assert.AreEqual("Customer", routeData.Values("controller"), "Customer controller expected");
    Assert.AreEqual("1", routeData.Values("id"), "Customer ID = 1 expected");
    Assert.AreEqual("Info", routeData.Values("action"), "Info action expected");
}
Dim routeData = GetRouteDataForUrl(routes, "~/Customer/Info/1")

Assert.IsNotNull(routeData, "Should have found the route")
Assert.AreEqual("Customer", routeData.Values("controller"), "Customer controller expected")
Assert.AreEqual("1", routeData.Values("id"), "Customer ID = 1 expected")
Assert.AreEqual("Info", routeData.Values("action"), "Info action expected")
End Sub

9. Implement a test method for the route 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:

a. A route has been found.

b. The controller that is called is the Address controller.

c. The Id parameter value is one.

d. The action method that is called is Create.

(Code Snippet – Intro to Asp.Net MVC Lab – ShouldAddressTakeNewRoute CSharp)

C#

[TestMethod]
public void ShouldAddressTakeCreateRoute()
{
    RouteCollection routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);

    RouteData routeData = GetRouteDataForUrl(routes, "~/Address/Create/1");

    Assert.IsNotNull(routeData, "Should have found the route");
    Assert.AreEqual("Address", routeData.Values["Controller"], "Address controller expected");
    Assert.AreEqual("1", routeData.Values["Id"], "Customer ID = 1 expected");
    Assert.AreEqual("Create", routeData.Values["action"], "Create action expected");
}

(Code Snippet – Intro to Asp.Net MVC Lab – ShouldAddressTakeNewRoute VB)

Visual Basic

<TestMethod()> _
Public Sub ShouldAddressTakeCreateRoute() _
    Dim routes As New RouteCollection()
}
MvcApplication.RegisterRoutes(routes)

Dim routeData = GetRouteDataForUrl(routes, "~/Address/Create/1")

Assert.IsNotNull(routeData, "Should have found the route")
Assert.AreEqual("Address", routeData.Values("Controller"), "Address controller expected")
Assert.AreEqual("1", routeData.Values("Id"), "Customer ID = 1 expected")
Assert.AreEqual("Create", routeData.Values("action"), "Create action expected")

End Sub

Note: If you want to test these methods run successfully you can now follow the steps in Verification 1.

Task 3 – Testing Customer Controller

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.

1. Import the CustomerController test class. To do this, right-click on the Controllers folder under the MvcSampleApp.Test project, point to Add / Existing Item. In the Add Existing Item dialog, browse to Source\Assets folder (choosing the folder that matches the language of your preference) and select CustomerControllerTest.cs (C#) or CustomerControllerTest.vb (Visual Basic.)

2. Import the stub version for the AdventureWorks repository. To do this, right-click on the MvcSampleApp.Test project, point to Add / Existing Item. In the Add Existing Item dialog, browse to Source\Assets folder (choosing the folder that matches the language of your preference) and select AdventureWorksRepositoryStub.cs (C#) or AdventureWorksRepositoryStub.vb (Visual Basic.)

Note: The AdventureWorksRepositoryStub class implements the IAdventureWorksRepository interface. You will include this interface in a further step.


4. Import the AdventureWorks repository interface. To do this, right-click on the Models folder under MvcSampleApp project, point to Add / Existing Item. In the Add Existing Item dialog,
browse to Source\Assets folder (choosing the folder that matches the language of your preference) and select IAdventureWorksRepository.cs (C#) or IAdventureWorksRepository.vb (Visual Basic.)

5. Modify the AdventureWorksRepository class to implement the AdventureWorks repository interface. To do this, open the AdventureWorksRepository.cs (C#) or AdventureWorksRepository.vb (Visual Basic) file under the Models folder in the MvcSampleApp project and replace the class declaration using the following code.

C#

```csharp
public class AdventureWorksRepository : IAdventureWorksRepository
```

Visual Basic

```vbnet
Public Class AdventureWorksRepository
    Implements IAdventureWorksRepository
```

**Note:** In Visual Basic you must explicitly implement the new interface by using the Implements keyword decorating all the necessary methods.

For instance, the GetCustomers methods must be decorated with Implements IAdventureWorksRepository.GetCustomers in order to implement the interface method.

6. 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 (C#) or CustomerController.vb (Visual Basic) file under the Controllers folder in theMvcSampleApp project and replace the repository field declaration with the following constructor methods.

C#

```csharp
public class CustomerController : Controller
{
    private IAdventureWorksRepository repository;

    public CustomerController()
    {
        this.repository = new AdventureWorksRepository();
    }

    public CustomerController(IAdventureWorksRepository repository)
    {
        this.repository = repository;
    }
    ...
```
Public Class CustomerController
    Private repository As IAdventureWorksRepository

    Public Sub New()
        Me.repository = New AdventureWorksRepository()
    End Sub

    Public Sub New(ByVal repository As IAdventureWorksRepository)
        Me.repository = repository
    End Sub

    ... .
   End Class

7. Open the CustomerControllerTest.cs (C#) or CustomerControllerTest.vb (Visual Basic) class file 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.

(Code Snippet – Intro to Asp.Net MVC Lab – ShouldLoadCustomerIndexView CSharp)

C#

    [TestMethod]
    public void ShouldLoadCustomerIndexView()
    {
        CustomerController controller = new CustomerController(new AdventureWorksRepositoryStub());
        ViewResult result = controller.Index(1) as ViewResult;

        Assert.IsTrue(string.IsNullOrEmpty(result.ViewName));
        Assert.IsNotNull(result.ViewData);
    }

(Code Snippet – Intro to Asp.Net MVC Lab – ShouldLoadCustomerIndexView VB)

Visual Basic

    <TestMethod()> _
    Public Sub ShouldLoadCustomerIndexView()
        Dim controller As New CustomerController(New AdventureWorksRepositoryStub())
        Dim result = TryCast(controller.Index(1), ViewResult)
8. 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.

(Codification Snippet – Intro to Asp.Net MVC Lab – ShouldLoadCustomerIndexPageView C#)

```csharp
[TestMethod]
public void ShouldLoadCustomerIndexPageView() {
    CustomerController controller = new CustomerController(new AdventureWorksRepositoryStub());
    ViewResult result = controller.Index(1) as ViewResult;
    Assert.IsInstanceOfType(result.ViewData.Model, typeof(CustomerViewData));
    CustomerViewData customerViewData = result.ViewData.Model as CustomerViewData;
    Assert.AreEqual(2, customerViewData.NextPage, "Page 2 expected");
    Assert.AreEqual(2, customerViewData.Customers.Count(), "2 Customers expected");
}
```

(Codification Snippet – Intro to Asp.Net MVC Lab – ShouldLoadCustomerIndexPageView VB)

```vbnet
<TestMethod()> _
Public Sub ShouldLoadCustomerIndexPageView()
```

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` (C#) or `AdventureWorksRepositoryStub.vb` (Visual Basic) file under the test project for implementation details.

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()`.
Dim controller As New CustomerController(New AdventureWorksRepositoryStub())
    Dim result = TryCast(controller.Index(1), ViewResult)
    Assert.IsTrue(string.IsNullOrEmpty(result.ViewName));
    Assert.IsNotNull(result.ViewData);

(Code Snippet – Intro to Asp.Net MVC Lab – ShouldLoadCustomerInfoView CSharp)

[TestMethod]
public void ShouldLoadCustomerInfoView()
{
    CustomerController controller = new CustomerController(new
    AdventureWorksRepositoryStub());
    ViewResult result = controller.Info(1) as ViewResult;
    Assert.IsTrue(string.IsNullOrEmpty(result.ViewName));
    Assert.IsNotNull(result.ViewData);
}

(CODE SNIPPET – Intro to Asp.Net MVC Lab – ShouldLoadCustomerInfoView VB)

<TestMethod()> _
Public Sub ShouldLoadCustomerInfoView()
    Dim controller As New CustomerController(New
    AdventureWorksRepositoryStub())
    Dim result = TryCast(controller.Index(1), ViewResult)
    Assert.IsTrue(String.IsNullOrEmpty(result.ViewName))
    Assert.IsNotNull(result.ViewData)
End Sub

10. 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.
Next Step

Exercise 3: Verification

In order to verify that you have correctly performed all steps of exercise three, proceed as follows:

Verification 1

In this verification you will make sure all the route tests implemented pass successfully.

1. Open the solution `MvcSampleApp.sln` in Visual Studio if not already opened.
2. Press `Ctrl + Shift + B` to build the solution.
3. In `Test` menu, point to `Windows` and select `Test List Editor`.
4. Select the following tests:
a. ShouldCustomerTakeDefaultRoute
b. ShouldCustomerTakeInfoRoute
c. ShouldAddressTakeCreateRoute

![Test List Editor](image)

**Figure 50**
Selecting tests to run

**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* (C#) or *AccountControllerTest.vb* (Visual Basic) file for detailed examples of controller unit testing.

5. In Test menu, point to Run and select **Tests in Current Context**.
6. Make sure that all tests pass.

![Test Results](image)

**Figure 51**
Tests pass
Verification 2

In this verification you will make sure that all the tests implemented for the **Customer** controller pass successfully.

1. Open the solution **MvcSampleApp.sln** in Visual Studio if not already opened.
2. Press **Ctrl + Shift + B** to build the solution.
3. In **Test** menu, point to **Windows** and select **Test List Editor**.
4. Select the following tests:
   a. ShouldLoadCustomerIndexView
   b. ShouldLoadCustomerIndexPageView
   c. ShouldLoadCustomerInfoView
   d. ShouldLoadCustomerOneInfoView

![Test List Editor](image)

**Figure 52**

*Selecting tests to run*

5. In **Test** menu, point to **Run** and select **Tests in Current Context**.
6. Make sure that all tests pass.

![Test Results](image)

**Figure 53**

*Tests pass*
Next Step

Summary

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.

Next Step

Appendix

Moq Library

It is a library that allows you, among other things, to create mock objects and set their expected behavior in an easy way.

The **GetRouteDataForUrl** method used in this exercise works in the following way.

### C#

```csharp
private static RouteData GetRouteDataForUrl(RouteCollection routes, string url)
{
    var mockRequest = new Mock<HttpRequestBase>();
    mockRequest.Setup(r => r.AppRelativeCurrentExecutionFilePath).Returns(url);
    mockRequest.Setup(r => r.PathInfo).Returns(String.Empty);
    var mockContext = new Mock<HttpContextBase>();
    mockContext.Setup(c => c.Request).Returns(mockRequest.Object);
    RouteData routeData = routes.GetRouteData(mockContext.Object);
    return routeData;
}
```

### Visual Basic

```vbnet
Private Function GetRouteDataForUrl(routes As RouteCollection, url As String) As RouteData
    Dim mockRequest As New Mock(Of HttpRequestBase)()
    mockRequest.Setup(Function(r) r.AppRelativeCurrentExecutionFilePath).Returns(url)
    mockRequest.Setup(Function(r) r.PathInfo).Returns(String.Empty)
    Dim mockContext As New Mock(Of HttpContextBase)()
    mockContext.Setup(Function(c) c.Request).Returns(mockRequest.Object)
    Dim routeData As RouteData = routes.GetRouteData(mockContext.Object)
    Return routeData
End Function
```
Private Shared Function GetRouteDataForUrl(ByVal routes As RouteCollection, ByVal url As String) As RouteData
    Dim mockRequest = New Mock(Of HttpRequestBase)()
    mockRequest.Setup(Function(r) r.AppRelativeCurrentExecutionFilePath).Returns(url)
    mockRequest.Setup(Function(r) r.PathInfo).Returns(String.Empty)

    Dim mockContext = New Mock(Of HttpContextBase)()
    mockContext.Setup(Function(c) c.Request).Returns(mockRequest.Object)
    Dim routeData As RouteData = routes.GetRouteData(mockContext.Object)
    Return routeData
End Function

1. The first thing is to create your mock object, which is done using the `Mock<T>` (C#) or `Mock(Of T)` (Visual Basic) class. You need to create a mock request object that will allow faking an HTTP request.

   **C#**
   var mockRequest = new Mock<HttpRequestBase>();

   **Visual Basic**
   Dim mockRequest = New Mock(Of HttpRequestBase)()

2. 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.

   **C#**
   mockRequest.Setup(r => r.AppSettings).Returns(url);
   mockRequest.Setup(r => r.PathInfo).Returns(String.Empty);

   **Visual Basic**
   mockRequest.Setup(Function(r) r.AppSettings).Returns(url)
   mockRequest.Setup(Function(r) r.PathInfo).Returns(String.Empty)

3. Now that you have the mock request configured, you need to create a mock HTTP context and assign the mock request to it.

   **C#**
   var mockContext = new Mock<HttpContextBase>();
4. Finally, with the mock context, you can request the route data from the passed in `RouteCollection` using the mock context.

**C#**

```csharp
RouteData routeData = routes.GetRouteData(mockContext.Object);
return routeData;
```

**Visual Basic**

```vbnet
Dim routeData As RouteData = routes.GetRouteData(mockContext.Object)
Return routeData
```