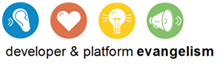
Demo Script

Introduction to ASP.NET MVC

Lab version: 1.0.0

Last updated: 3/13/2009



1. Contents

[Overview 3](#_Toc224724203)

[Key Messages 3](#_Toc224724204)

[Demo Diagram 4](#_Toc224724205)

[Key Technologies 4](#_Toc224724206)

[Time Estimates 5](#_Toc224724207)

[Setup and Configuration 5](#_Toc224724208)

[Task 1 – Running the Configuration Wizard 5](#_Toc224724209)

[Demo Flow 6](#_Toc224724210)

[Opening Statement 7](#_Toc224724211)

[Step-by-Step Walkthrough 8](#_Toc224724212)

[Segment #1 – Creating and Exploring an MVC Application 9](#_Toc224724213)

[Segment #2 – Developing an MVC Application 20](#_Toc224724214)

[Segment #3 – Unit Testing an MVC Application 28](#_Toc224724215)

[Summary 33](#_Toc224724216)



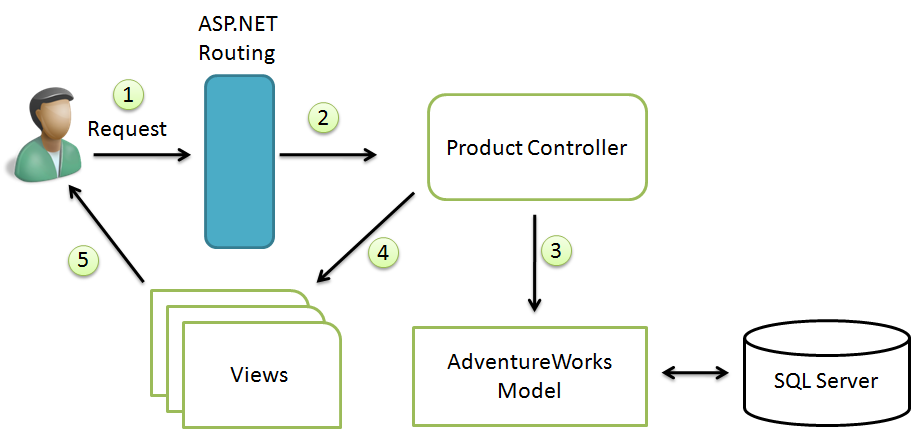
Overview

* 1. This document provides setup documentation, step-by-step instructions, and a written script for showing an how to build an ASP.NET MVC application, its structure and how to unit test the controllers and the routing. This document can also serve as a tutorial or walkthrough of the technologies being exposed.
  2. This demo is aimed to show the basics and capabilities of ASP.NET MVC, their components and how they interact. It also shows how to create views and controllers using the provided scaffolding. Finally the demo shows how to test an ASP.NET MVC application.

### Key Messages

* 1. You will see how to create an ASP.NET MVC application from scratch using the Visual Studio tooling to easily create controllers and views, understanting an MVC application structure and each one of its components.
  2. You will see how to unit test ASP.NET MVC applications, specifically how to create unit tests for the controller actions and how to test routes.

### Demo Diagram

* 1. 

### Key Technologies

This demo uses the following technologies:

* 1. ASP.NET MVC
  2. ASP.NET Routing
  3. Microsoft SQL Server 2008 or Microsoft SQL Server 2005

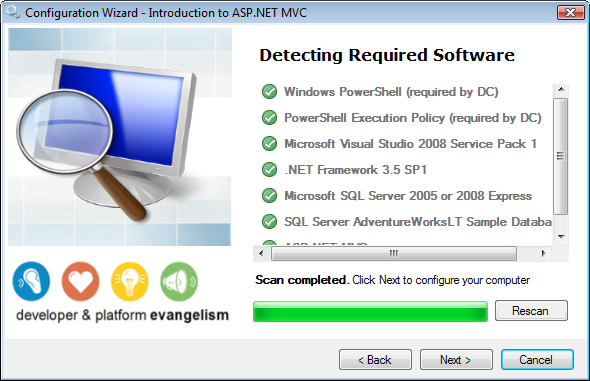
### Time Estimates

* + Estimated time for setting up and configuring the demo: 5 min
  + Estimated time to complete the demo: 60 min

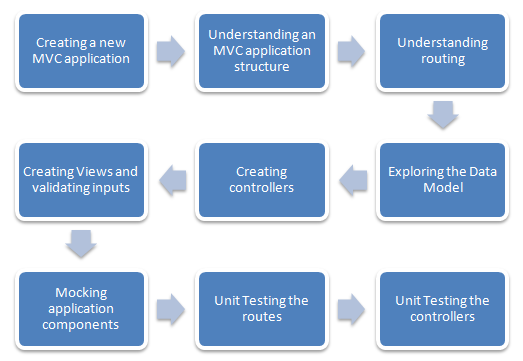
Setup and Configuration

The setup and configuration for this demo involves running the Configuration Wizard tool included with the training kit.

Task 1 – Running the Configuration Wizard

* 1. The following steps describe how to run the Configuration Wizard tool included with the demo to verify that you have the prerequisite components properly installed.
  2. Run the **CheckDependencies.cmd** command script located under the **..\Demos\introToAspNetMvc\setup** if you have not done it previously. This script will launch the Configuration Wizard for the Training kit. The Configuration Wizard is designed to check your computer to ensure that it is properly configured with all the dependencies required to use the demo.
  3. Click through the steps in the Configuration Wizard to get to the Required Software step when a scan of your computer will be performed. If you do not have the necessary dependencies, then install them using the links provided by the tool and rescan your computer.
     1. 
  4. Once the required software is properly installed click **Next** to configure your computer for this demo. A set of scripts will be executed to create a database alias required by the demo.

Demo Flow

1. The following diagram illustrates the high-level flow for this demo and the steps involved:
2. 

Opening Statement

* 1. Today I would like to walk you through a demo application built to show the ASP.NET MVC basics, its capabilities, and its advantages.
  2. 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.

This demo uses the AdventureWorksLT Sample Database as the data source. The application displays a list of customers; when you click one, its details are displayed including its registered addresses. Each customer can have more than one registered address. The application allows you to perform CRUD operations on the customer’s addresses.

* 1. Is important to notice that the **ASP.NET MVC** framework provides an alternative to the ASP.NET Web-forms pattern for creating MVC-based Web applications. It is an optional offering and does not replace the existing WebForms Page Controller model. You can even build a single application and have parts of it written using WebForms and parts written using an MVC approach.

In this demo you will specifically see three things:

* 1. You will see how to create a sample MVC application from scratch, understanding the MVC pattern basics and the solution structure.
  2. You will see how to build the MVC application, that is creating the model, the controllers and the views.
  3. You will see how to unit test MVC applications, mocking the application elemnts, testing the controllers and routes.
  4. **Note:** For more information about the MVC (Model View Controller) architectural pattern and the ASP.NET MVC framework, see the Introduction to ASP.NET MVC Lab.

Step-by-Step Walkthrough

1. This demo is composed of the following segments:
   1. Creating and Exploring an MVC Application
   2. Developing an MVC Application
   3. Unit Testing an MVC Application

### Segment #1 – Creating and Exploring an MVC Application

|  |  |  |
| --- | --- | --- |
| Action | Script | Screenshot |
| 1. Open Microsoft Visual Studio 2008 from **Start** | **All Programs** | **Microsoft Visual Studio 2008** **| Microsoft Visual Studio 2008**.  2. On the **File** menu, point to **New**, and click **Project**.  3. 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. Provide the desired location.  4. Change the Name to **MvcSampleApp** and click **OK**. | * + Let’s open Visual Studio 2008, and create a new MVC Application from scratch using the built-in Visual Studio **ASP.NET MVC Web Application** template included in the MVC Framework 1.0.   + This template creates by default the entire solution structure and conventions that best fits an ASP.NET MVC application.   + 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 Application with the default project structure and conventions where you can implement your MVC application. The second project is a testing project where you can write unit tests for your MVC components.   + In a moment you will see how the project is organized in terms of folders, and which files should be placed inside each of them.   + The **Test framework** drop-down list on the **Create Unit Test Project** dialog has currently one option for **Visual Studio Unit Test**. This list is extensible and may include other testing framework options if 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. Press **Ctrl+ Alt + L** to see the **Solution Explorer** and expand the folders to expose its content.  6. Show the different folders in the solution with its contents. | * + When you create an ASP.NET MVC Web application project, MVC components are separated into 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**: This 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:** In this folder you can place Controller classes. In an MVC based application, these are the components responsible for handling end user interaction, manipulating the model, and ultimately choosing a view to render to display UI.     - **Models**: In this folder you can place classes that represent the application model of 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**: In this folder you can place the JavaScript files of your application.     - **Views**: This folder is the recommended location for placing 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 that 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.     - **Views\Shared:** This folder 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. |  |
| 7. Open the **Home Controller**. To do this, in **Solution Explorer**, double-click the **HomeController.cs** file under **Controllers** folder.  8. Show how **HomeController** inherits from the **Controller** class, and highlight the methods representing action methods. | * + Before running the application, you should have a basic notion about how the framework works behind the scenes. We will start by understanding what is a controller and how it works in ASP.NET.   + 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.   + Let’s move to see more concretely what controllers are about. I will open the home controller to show how a Controller class is defined, and how action methods are created to handle user interaction. 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.   + The action methods names bind to actions in the URL request.  In order for an action method to be callable, it must be public, and do not have a **[NonActionAttribute]** associated to it.   + 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**.   + 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.   + 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, a downloadable file, a JavaScript file, and more. |  |
| 9. Open the **Index** view associated to the **HomeController**. To do this, in **Solution Explorer**, double-click **Index.aspx** under **Views\Home** folder.  10. Show that the View inherits from **System.Web.Mvc.ViewPage**.  11. Show the **Html.Enconde** call. | * + Let’s see how is that a view can be associated to the action method of a controller. As seen before, all the views corresponding to a particular controller are located under the **Views\Controller** folder. In this case, I want to open the **Index** view of the **Home** controller, which will be rendered whenever the **Index** action method is called.   + 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.   + Views should not contain any application logic or database retrieval code. Views render the appropriate UI by using this **ViewData** class, which is an MVC view-related data object provided by the controller.   + 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>**, **T** indicates the data object type of the strongly typed **ViewData** provided by the controller.   + 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.   + The **Html** object used in this code snippet 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. |  |
| 12. Double-click **Global.asax** in **Solution Explorer** to see how the preconfigured routes are defined.  13. Show the default route in the **RegisterRoutes** method. | * + Until now, we have seen controllers, their associated views, and how they are rendered based on the action method that handles the request. But there is an important piece missing: How does the framework knows which controller should instantiate, and which action method should invoke based on the request URL?   + The answer is **ASP.NET Routing**. This mechanism parses variables in the URL according to a pattern that you define, and automatically passes the variables to a controller action as parameter arguments. With this approach, the URLs do not have to map to specific files in a Web site.   + These patterns or rules are specified in the ASP.NET **Application** class of the **Global.asax** file. Let’s open the **Global.asax** file, and see that by default **ASP.NET MVC** projects provide a preconfigured set of URL routing rules that enable you to easily get started on an application without needing to explicitly configure anything. However, you can define new routing rules that better fits your scenario in the **RegisterRoutes** method in **Global.asax**.   + 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 the id which are the parameters that should be passed in.   + The third parameter in the current **MapRoute** overload we are usingspecifies the action that should be performed if the request URL does not match with the registered route. This action will be to instantiate the **Home** controller, and invoke the **Index** action method. We will see how this works in a few minutes when browsing the default page. |  |
| 14. 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**.  15. If the **Script Debugging Disabled** dialogappears, click **Yes** to continue debugging.  16. Show the URL in the browser: [**http://localhost:50000**](http://localhost:50000). | * + It’s time to see our MVC application working. Let’s start a new instance and see what is happening with the default page.   + A request is made to [**http://localhost:50000**](http://localhost:50000) which is intercepted by the **ASP.NET Routing** engine applying the ***Default*** registered route we’ve seen before ***[controller]/[action]/[id]***. Since the URL does not contain any controller to map, the routing engine instantiates the **Home** controller, and invokes the **Index** action method as seen before. |  |
| 17. With the solution running, open the **HomeController.cs** file for debugging.  18. Insert a breakpoint in the **return View()** line of the the **About** action method. To do this, set the cursor in the line and press **F9**.  19. After the breakpoint is hit, press **F5** to continue execution. | * + Before browsing the **About** link I will put a breakpoint in the **About** action method to verify that is indeed handling the request for the current route.   + Let’s click the **About** link and see how the breakpoint is hit. I will resume the execution and the view for the **About** action will be rendered.   + Pay attention to the URL in the **About** page: [**http://localhost:50000/Home/About**](http://localhost:50000/Home/About)**.** According to default defined route, **ASP.NET** **Routing** understands that has to instantiate a **Home** controller, and invoke the **About** action method with no additional parameters. |  |

### Segment #2 – Developing an MVC Application

|  |  |  |
| --- | --- | --- |
| Action | Script | Screenshot |
| 20. Open the solution file **MvcSampleApp.sln** located under **..\Demos\introToAspNetMvc\code\**.  21. Open the **AdventureWorks** entity data model, located in the **Model** folder.  22. Show the generated model created with ADO.NET Entity framework, which contains only the **Products** table. | * + Let’s open a solution that has already some controllers, views and the model implemented. As we talked before, these are the principal actors of an MVC application.   + It is now time to introduce the model. These components of an application are responsible for maintaining state; in web applications this state is typically persisted inside a database.   + The ASP.NET MVC Framework enables you to use any data access pattern or framework you want in order to retrieve and manage your models.You can use ADO.NET DataSets/DataReaders if you want. You can also use object relational mapper (ORM) like ADO.NET Entity framework, NHibernate, LLBLGen, if you prefer.   + In this case the model was created using ADO.NET Entity framework.   + Let’s open the model which is located in the **Model** folder. As you can see, our model is very simple and only contains a products table. |  |
| 23. Open the **AdventureWorksRepository** class, located in the **Model** folder.  24. Show the different methods of the class that perform the CRUD operations on the **Product** table of the database. | * + In the **Model** folder, we have an additional class, the **AdventureWorksRepository**. The repository exposes methods to retrieve entities from the model, generating a level of abstraction from the underlying data model.   + As you can see, the **AdventureWorksRepository** class contains methods to query the database and perform CRUD operations on the database. The **GetProducts** and **GetProductsByID** retrieve data from the database, while **AddProduct**, **UpdateProduct**, and **DeleteProduct** methods; insert, update, and delete products from the database, respectively. |  |
| 25. Create an empty MVC controller class. In the **Solution Explorer** right-click the **Controllers** folder, point to **Add** and select Controller.  26. Leave the **Controller Name** as it is and select the **Add action methods for Create, Update and Details scenarios**, and then click **Add**.  27. In the recently created controller, show the automatically created stub methods. | * + We will create a controller using the Visual Studio tooling. To do this, right-click the **Controllers** folder, point to **Add** and click **Controller**.   + As I mentioned before, the convention for controller naming is to add the **Controller** suffix to the class name; here you can see how this convention is integrated in the **Add Controller** dialog window. We can also select the option to add action methods for Create, Update and Details scenarios automatically. Clicking the **Add** button will cause the controller class to be created and added to the project.   + This controller class will contain stub action methods for the selected scenarios that the MVC framework has automatically created. You only have to complete these methods with your own logic. For the **Edit** and **Create** scenarios two action methods were created, one for **GET** requests and one for **POST** requests. |  |
| 28. Open the **ProductsController** class, located in the **Controller** folder.  29. Show the implemented **Index** action method of this controller. | * + For practical purposes, I will be working with the existing **Product** controller that has some of its action methods already implemented. Let’s see the **Index** method.   + In the **Index** method we will create a new instance of the **ProductViewData** and set its properties: a list of all products to show, and the previous and next page number required to implement the list pagination.   + Finally, we return the **Index** view, passing the view data as a parameter. |  |
| 30. Show both **Create** action methods of this controller. | * + Let’s see the Create action methods used in Insertion scenarios. As you can see, there are two action methods for this, one to respond to **GET** requests (where the initial empty form will be displayed) and one for **POST** requests (to impact the insertion in the model and the database).   + Notice that the method used for **POST** requests has the **AcceptVerbs** attribute applied. This method creates a new product instance, use the **UpdateModel** helper method to bind the form data to the instance properties; and finally use the repository class to insert the product in the database. |  |
| 31. Open the **ProductsViewData** class, located in the **ViewData** folder.  32. Show the two properties of the class that are used for rendering the product controller’s views. | * + Now, let’s examine the **ProductViewData** class. This class contains the ViewData for some of the Products’ views.   + The class contains the **Products**, **PreviousPage**, and **NextPage** properties, which are used for rendering some of the views of the product controller. |  |
| 33. Open the **Default1Controller** controller class created previously.  34. Right-click inside the **Index** method’s code and click **Add View**. In the dialog window, select **Create a strongly-typed view**. In the **ViewData class** and **View content** combo boxes, select **MvcSampleApp.ViewData.ProductViewData** and **List** respectively.  35. Repeat the same step for the **Edit** action method, selecting **MvcSampleApp.Models.Product** as the **ViewData class** and **Edit** as the **View content**.  36. Open the Solution Explorer window, and show the folder structure of the solution created by the scaffolding. | * + Now we will create the views for each controller’s action method; we will use the **Default1Controller** that we have created previously. When you right-click inside the code of an action method, the **Add View** option appears. This option allows you to create empty view templates, or automatically generated view templates based on the type of object passed to the view by the Controller action method. The scaffolding mechanism uses reflection when creating view templates passed to it.   + I will create a view based on the **Index** and **Edit** action methods of the **Product** controller, providing the **ProductViewData** and **Product** view data classes, and using the **List** and **Edit** view contents respectively.   + Let’s see the solution explorer. Note that the framework created the new view and a folder for the controller inside the **Views** folder. |  |
| 37. Open the **Edit.aspx** page, located in the **Views\Product** folder.  38. Show the design view of this page.  39. Open the **Index.aspx** page, located in the **Views\Product** folder.  40. Show the design view of this page and the HTML code. | * + For practical purposes, we will see the existing scaffolded views of the product controller. I will open the **Edit** view. You can see that the framework has created the edit form automatically. These views work very well, but typically need some work to adjust them to your needs. For example, in this view I deleted the **ProductId** field because it should not an editable field.   + For instance, the generated **Index** view took more effort to adjust it to the desired state, because we had to manually implement the pagination and select which field will be shown in the list. |  |
| 41. Open the **ProductsController** class, located in the **Controller** folder.  42. Set a breakpoint in the **Index** action method of this controller. Run the application and click the Products link. | * + Now we will see how the action method is invoked when a request is made to **/Product/Index**. To do this, we will set a breakpoint in the Index action method of the products controller.   + Let’s run the application and go to the Products list. The application execution breaks and we confirm that the routing is working well. Notice that by default the Index method is set as the default method of every controller. That is why when the URL points to a controller and no action method is specified, the default action method (**Index**) will be invoked. |  |
| 43. Open the **Create.aspx** page and show the validation logic inside the HTML code.  44. Run the application. Go to the Products page and click **Create New**.  45. In the form, delete the **ProductID** field, enter some input in the **Name** field, enter some characters in the **StandardCost** field, and click **Save**. | * + Finally, we will see how the default form validation works.   + Let’s open the **Create.aspx** page to see the validation code that is inserted automatically by the MVC framework. You can see that for every form field, the **ValidationMessage** helper method was added. By default an “\*” will be shown when an error occurs on the field, but this message can be customized. At the top of the form the **ValidationSummary** helper method was also added to show a list of all errors in the form. Keep in mind that this provided validation works with the constraints defined in the model.   + So let’s run the application and we will try to create a new product. We will leave the **ProductID** field empty, enter something in the **Name** field, and enter some characters in the **StandardCost** field. Now let’s click save. The validation message will be shown and the fields with invalid inputs will be highlighted. Notice that the input entered in the form is not cleared, its state is preserved. This is the expected behavior of web application and saves the user for re-entering the data manually. |  |

### Segment #3 – Unit Testing an MVC Application

|  |  |  |
| --- | --- | --- |
| Action | Script | Screenshot |
| 46. Expand the **MvcSampleApp.Tests** project.  47. Expand the **Controllers** folder, and show all the files contained in the test project. | * + Now we will focus our attention in the test project. In this project you can find all the components to make unit testing to your 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.   + The **Controllers** folder contains all the classes that tests our controllers. As you can see here, we have one **TestClass** per controller. Each class will have all the methods that tests specific controller behavior.   + We have also some other files such as **RouteTests** that contains all tests related to routing in our Mvc Application. |  |
| 48. Open the **RouteTest.cs** file.  49. Show the **GetRouteDataForUrl()** method. | * + Let’s see how can we test the route matching rules defined in **Global.asax**.   + **GetRouteDataForUrl** method emulates the mechanism that performs the ASP.NET Routing engine when it receives an URL, and returns the matching route data from the loaded application route collection. This method uses **moq** mocking library to mock the **HttpContextBase** classwhich enables to find the matching **RouteData** in the **RouteCollection** basedonthe URL parameter. |  |
| 50. Show the **ShouldProductTakeInfoRoute()** method.  51. Insert a breakpoint in the first line of **ShouldProductTakeInfoRoute()** method. To do this, set the cursor on the line and press **F9**.  52. Leave the cursor inside the method. In **Test** menu, point to **Debug,** and select **Tests in Current Context**.  53. Keep pressing **F10** to **step over** to the next line until the test pass. | * + Let’s pick one routing test method to see how this testing works.   + **ShouldProductTakeInfoRoute** is testing that when we make a request to **~/Product/Info/1**, the default registered route establishes that **Product** is the matching controller, **Info** is matching view, and **1** is the matching parameter.   + Let’s analyze this step by step. I will insert a breakpoint in the first line and step over until the test pass.   + First, we create an empty **RouteCollection**. We fill this collection by calling the **RegisterRoutes()** method defined in **Global.asax**. Consequently, the **RouteCollection** will contain the default route.   + We call the **GetRouteDataForUrl()** to obtain the concrete route that matches our URL. Remember that this method mocks the ASP.NET routing behavior.   + Once we have the concrete route, we must make sure that it is pointing to the right controller, views and parameters. This is what the following **Assert**s are doing.   + Every **Assert** is correct**,** so we can see how the test passes. |  |
| 54. Open the **ProductControllerTest.cs** file under the **Controllers** folder.  55. Show the test method signatures.  56. Open the **AdventureWorksRepositoryStub.cs** file, and make a quick review to its methods. | * + Now it is time to test some controllers. More concretely, we will see how to test some action methods of the **Product** controller.   + In the application, the **Product** controller uses **AdventureWorks** entity data model to interact with the database. The idea here is to also simulate this behavior by creating a stub class of the **AdventureWorksRepository** so as to have full control over these tests. If any of them fail, you will be sure that the defect was produced within the **Controller** component rather than in the **AdventureWorks** entity data model.   + **GetProducts()** method returns just two fixed Products. While **GetProductById()** method creates an empty **Product** and assigns the productId that comes as a parameter. |  |
| 57. Switch to **ProductControllerTest.cs** file.  58. Show the **ShouldLoadProductDetailsView()** method.  59. Insert a breakpoint in the first line of **ShouldLoadProductDetailsView()** method. To do this, set the cursor on the line and press **F9**.  60. Leave the cursor inside the method. In **Test** menu, point to **Debug,** and select **Tests in Current Context**.  61. Keep pressing **F10** to **step over** to the next line until the test pass. | * + Let’s pick one of the **Product** test method to see how this testing works.   + **ShouldLoadProductDetailsView** istesting thatthe **Details** action method is obtaining the specified **Product** from the repository, and returning the correct **ViewData** type having been filled with the proper content.   + Let’s analyze this step by step. I will insert a breakpoint in the first line and step over until the test pass.   + First, we create a new instance of the **ProductController** class, by passing the **AdventureWorksRepositoryStub**.By doing this we are telling the controller to use the stubbed repository, instead of the one created with entity data model.   + We call the **Details** action method passing the product id **609** as parameter. This method should return a **ViewResult**.   + Once we have the **ViewResult**, we must make sure that the instance is of type **Product**, and that contains a product with id **609**. This is what the following **Assert**s are doing.   + Every **Assert** is correct**,** so we can see how the test passes. |  |

Summary

* 1. In this demo, you have learnt which the core elements of an MVC application are, and how they interact. You have seen how to create an MVC application and how to use the scaffolding provided by the framework to create controllers and views. You have also seen how to unit test an ASP.NET MVC Application, specifically its routes and the controllers.
  2. **Feedback**
  3. What do you think of this demo?  What do you think of this training kit?  Your feedback is important; it will help us build the best products for you.  Please take a moment to provide it.