

Developing ASP.NET MVC Web Applications

Are you registered with
Onlinevarsity.com?

Yes



No



Did you download this book
from **Onlinevarsity.com?**

Yes



No



Scores

For each **YES** you score **50**

For each **NO** you score **0**

If you score less than 100 this book is illegal.

Register on **www.onlinevarsity.com**

Developing ASP.NET MVC Web Applications

© 2014 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2014



Dear Learner,

We congratulate you on your decision to pursue an Aptech Worldwide course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc.* These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

➤ Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of Web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

➤ Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

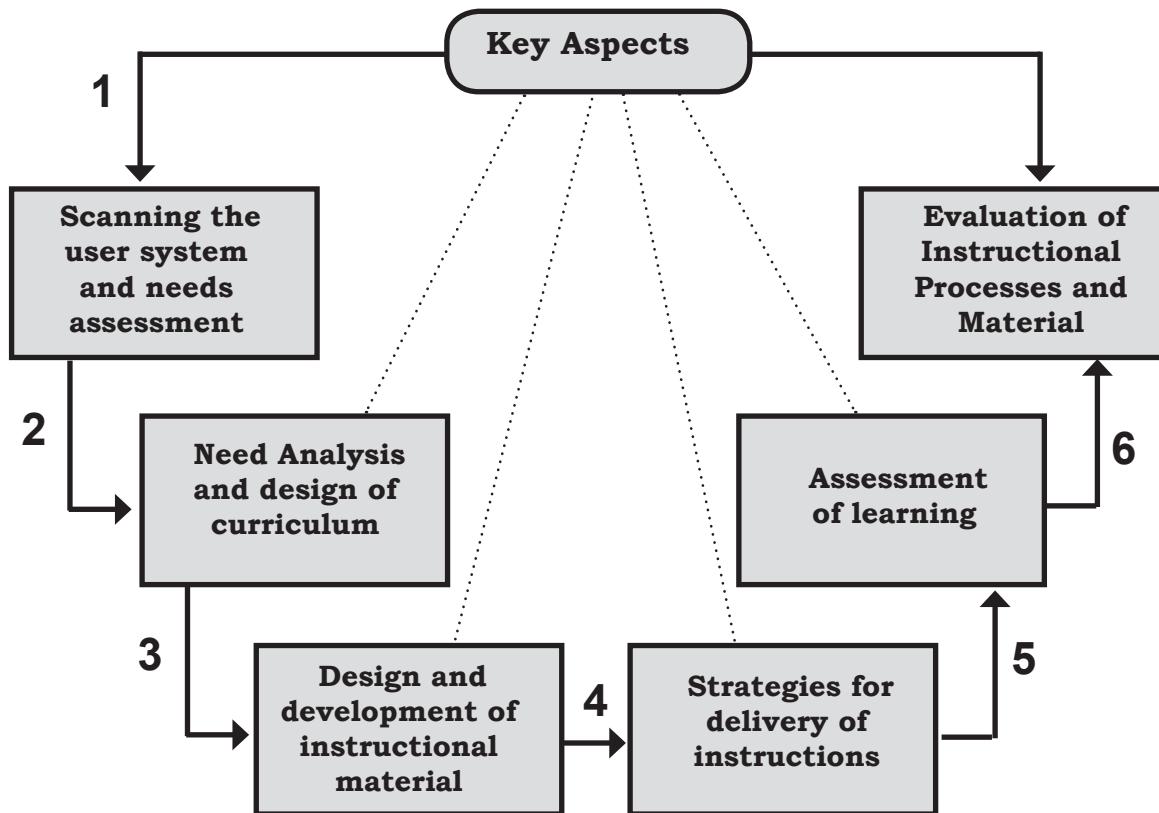
➤ Evaluation of instructional process and instructional materials

The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model





To enhance your knowledge,
visit the **REFERENCES** page





Preface

ASP.Net is a platform for developing Web applications that provide a comprehensive software infrastructure, a programming model, and a number of services required to develop robust Web application for PC, as well as mobile devices. This book will teach to develop advanced ASP.NET MVC applications using .NET Framework 4.5 and design the architecture of Web application and create MVC Models.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION



Table of Contents

Sessions

1. Introduction to ASP.NET MVC
2. Controllers in ASP.NET MVC
3. Views in ASP.NET MVC
4. Models in ASP.NET MVC
5. Data Validation and Annotation
6. Data Access
7. Consistent Styles and Layouts
8. Responsive Pages
9. State Management and Optimization
10. Authentication and Authorization
11. Security
12. Globalization
13. Debugging and Monitoring
14. Advanced Concepts of ASP.NET MVC
15. Testing and Deploying



Visit the
Frequently Asked Questions
section @

Session - 1

Introduction to ASP.NET MVC

Welcome to the Session, **Introduction to ASP.NET MVC**.

Web applications have revolutionized the way business is conducted. These applications enable organizations to share and access information from anywhere and at any time. Web applications have evolved from traditional Web application that served static content to Web applications that are dynamic and responsive. To create such dynamic and responsive Web application, you can use ASP.NET MVC.

ASP.NET MVC 5 has evolved from traditional ASP.NET Web pages and ASP.NET Web Form to provide an implementation of the Model View Controller (MVC) design pattern.

Visual Studio 2013 provides support to create ASP.NET MVC 5 applications. When you create an ASP.NET MVC 5 application in the Visual Studio 2013 IDE, the IDE automatically creates the required application directory structure and the minimal files required to run the application.

In this Session, you will learn to:

- ➔ Define and describe the layers of Web application
- ➔ Explain the structure of an ASP.NET MVC application
- ➔ Explain the evaluation of Web application
- ➔ Explain and describe how to create Web application in Visual Studio 2013

1.1 Overview of Web Application Development

Web applications are programs that are executed on a Web server and accessed from a Web browser. These Web applications allow you to share and access information over the Internet that can be accessed globally at any time. In addition, you can create Web applications for performing commercial transactions, such as buying or selling products in an online store. This type of Web application that implements such commercial transactions is known as E-commerce application.

A Web application is composed of separate layers. The layers that you need to implement in a Web application depends on the application requirements. The different Web application layers are discussed in the following topics.

1.1.1 Web Application Layers

Web applications are typically divided into three layers, where each layer performs different functionalities. The three layers of a Web application are as follows:

- Presentation layer
- Business logic layer
- Data layer

Figure 1.1 shows the three layers of a Web application.

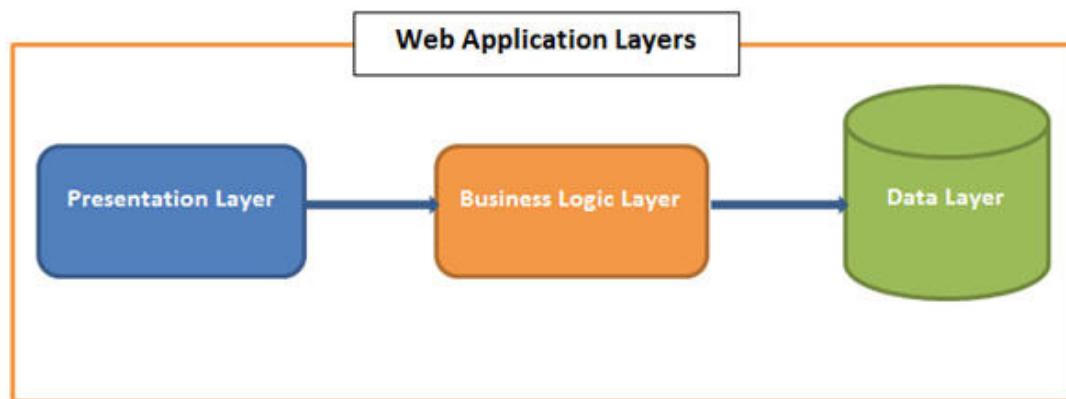


Figure 1.1: Three Layers of Web Application

In this figure, the Web application has the following three layers:

- **Presentation layer:** Enable users to interact with the application.
- **Business logic layer:** Enables controlling the flow of execution and communication between the presentation layer and data layer.

- **Data layer:** Enables providing the application data stored in databases to the business logic layer.

1.1.2 Web Application Architectures

The architecture of an application depends on the system in which the layers of the application are distributed and communicated to each other. Most of the applications are built by using these layers.

An application can be based on one of the following types of architectures:

- **Single-Tier Architecture:** In this architecture, all the three layers are integrated together and installed on a single computer.
- **Two-Tier Architecture:** In this architecture, the three layers are distributed over two tiers, a client and a server. The presentation layer resides on each client computer, the business logic layer resides either on the client or on the server, and the data access layer resides on the server.
- **Three-Tier Architecture:** In this architecture, the three layers of the application are distributed across different computers. Each layer communicates with the other layers with the help of a request-response mechanism.
- **N-Tier Architecture:** In this architecture, the components of the three-tier are further separated. For example, any business logic that needs to be present in one of the layers is separated from that layer to enable increase in performance and scalability.

1.1.3 Types of Web Pages

A Web application consists of Web pages. Web pages can be of the following types:

- **Static Web page:** Is a Web page that consists of only Hyper Text Markup Language (HTML) is a static Web page. A static Web page only presents content to users. Such Web page cannot respond to user actions.
- **Dynamic Web page:** Is a Web page that can respond to user actions. For example, when you type some text to search in www.google.com and click the **Search** button, the Web page responds by displaying your search results. This is an example of a dynamic Web page. You can create a dynamic Web page using HTML pages in combination with server-side and client-side scripts.

Note - Server-side scripts execute on a Web server to provide users with dynamic content that is based on the information generated through programming logic or retrieved from a database. Server-side scripts are written in server-side scripting languages, such as ASP.NET.

Note - Client-side scripts execute on the browser without interacting with the Web server that hosts the application. Client-side script can also provide users with dynamic content based on the information generated through programming logic that executes on the browser. Client-side scripts are written in client-side scripting languages, such as JavaScript.

1.2 Evolution of ASP.NET MVC

ASP.NET MVC is a framework for developing dynamic Web applications using the .NET Framework. Prior to ASP.NET MVC, dynamic Web applications based on the .NET Framework were developed using ASP.NET Web pages and ASP.NET Web Forms. The following topics discuss about these technologies before explaining how ASP.NET MVC helps in creating more robust and scalable Web applications.

1.2.1 ASP.NET Applications

ASP.NET is a server-side technology that enables you to create dynamic Web applications using advanced features, such as simplicity, security, and scalability, which are based on the .NET Framework.

ASP.NET applications are comprises the .aspx Web pages that combine both client-side and server-side scripts to build dynamic Web sites.

Once you create an ASP.NET application, you need to deploy the application on a Web server such as Internet Information Services (IIS) server, which is the Web server for the Windows platform. The request-response flows for an ASP.NET Web page comprises the following steps:

1. Browser sends a request for an ASP.NET Web page.
2. When the request arrives, the IIS server intercepts the request, loads the requested file, and forwards it to the ASP.NET Runtime for processing.
3. The ASP.NET Runtime that contains the ASP.NET script engine processes the requested ASP.NET page and generates the response.
4. The IIS server sends back the response to the Web server that requested the page.

Figure 1.2 illustrates the preceding steps.

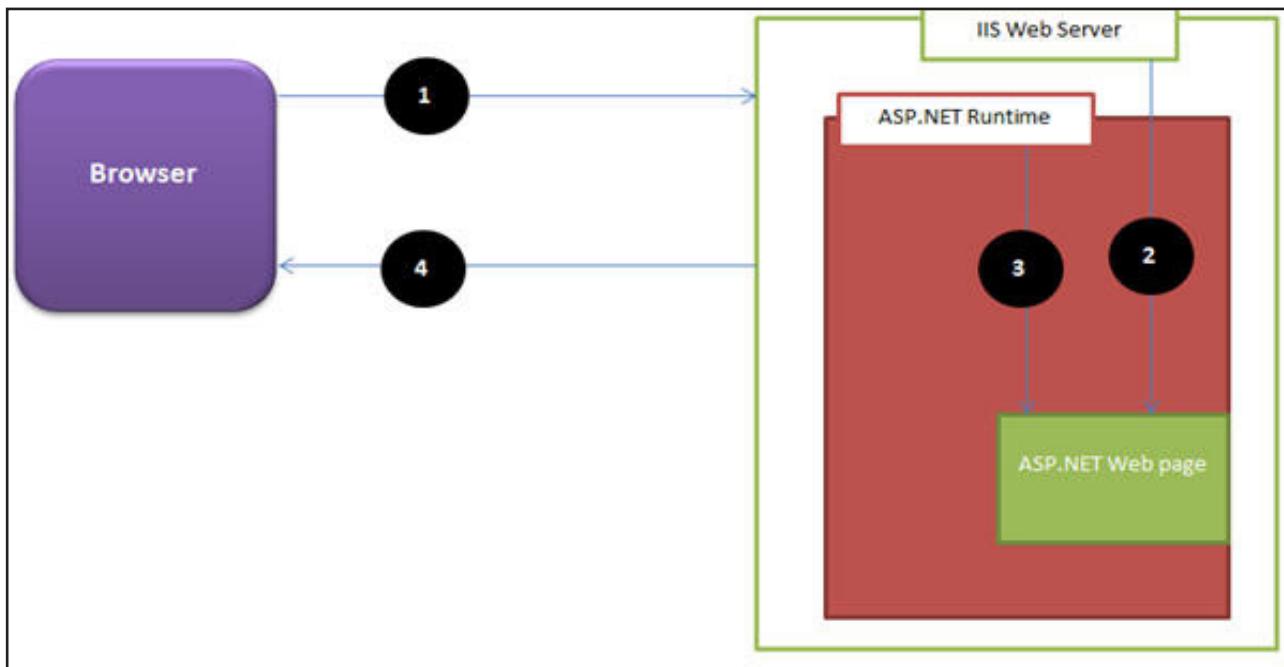


Figure 1.2: Processing of a Request for an ASP.NET File

1.2.2 ASP.NET Web Forms

The traditional ASP.NET Web applications gradually evolved to ASP.NET Web Forms to simplify development of dynamic Web applications. ASP.NET Web Forms introduced several User Interface (UI) controls that you can drag and drop to design your UI. Some examples of such UI controls are labels, text boxes, radio buttons, and check boxes. Once you drag and drop a UI control in your UI, you can easily set the properties, methods, and events for the control or the form. This enables you to specify how the form and its control should respond at runtime. Similar to traditional ASP.NET Web pages, Web Forms are written by using a combination of HTML, server controls, server code, and users request them through their browsers. It separates the HTML code from the application logic. To write server code for developing the logic for the page, you can use a .NET language, such as Visual Basic or C#.

Using Web Forms does not require you to have a hardcore developer background. You just need to be familiar with the user interface controls and event handling. Moreover, Web Forms allow a developer to use CSS, generate semantically correct markup, and handle the development environment created for HTML elements easily. This is because the developers just need to drag and drop the server controls and set their properties for designing the page. The markup of these controls is generated automatically.

1.2.3 ASP.NET MVC

ASP.NET MVC is based on the MVC design pattern that allows you to develop software solutions. For this, the MVC pattern provides a reusable solution to resolve common problems that occurs while developing a Web application.

You can use the MVC pattern to develop Web application with loosely coupled components. It is very difficult to manage Web applications that contain tightly coupled components. This is because, updating one component also requires updating the other components. To overcome such problems, you can use the MVC design pattern that enables separating data access, business, and presentation logic from each other.

While using the MVC design pattern, a Web application can be divided in the following three types:

- **Model:** Represents information about a domain that can be the application data of a Web application. In ASP.NET MVC applications, the model class represents this model.
- **View:** Represents the presentation logic to provide the data of the model. There can be multiple views for the same model. In ASP.NET MVC application, the files within the **View** folder of the application directory represent the view.
- **Controller:** Represents the logic responsible for coordinating between the view and model classes. The controller classes in an ASP.NET MVC application handles events thrown by the view and calls the corresponding model to be processed.

Figure 1.3 shows the communications between the model, view, and controller components.

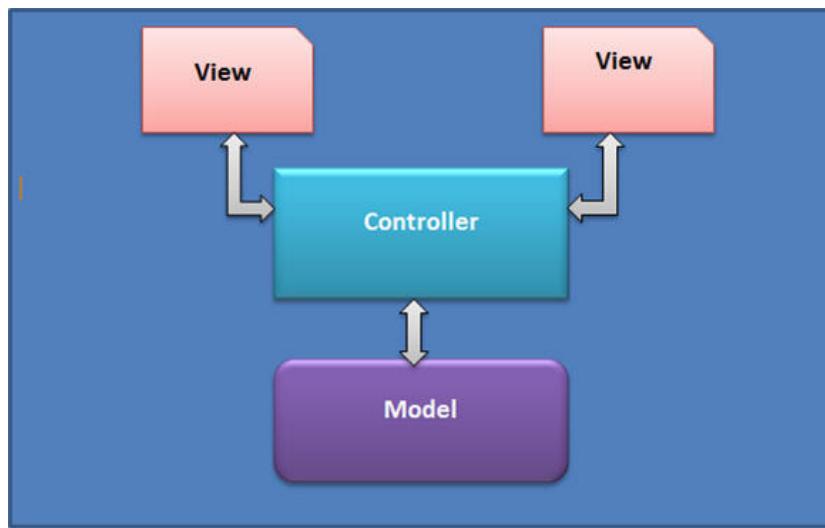


Figure 1.3: Communication between the Components of the MVC Pattern

As ASP.NET MVC is based on the MVC design pattern, it helps in developing applications in a loosely coupled manner and provides the following benefits:

- **Separation of concerns:** Enables you to ensure that various application concerns into different and independent software components. Thus, it allows you to work on a single component independently.
- **Simplified testing and maintenance:** Enables you to test each component independently. This helps you in ensuring that it is working as per the requirement of the application and then,

simplifies the process of testing, maintenance, and troubleshooting procedures.

- **Extensibility:** Enables the model to include a set of independent components that you can easily modify or replace based on the application requirement. Modifying or replacing these components does not affect the functionality of the application.

1.2.4 History of MVC

Following are the versions that depict the history of MVC:

→ **ASP.NET MVC 1**

This is the first version of ASP.NET MVC. ASP.NET MVC 1 was released on March 13, 2009 and targets .Net Framework 3.5. Visual Studio 2008 and Visual Studio 2008 SP1 provide support to develop ASP.NET MVC 1 applications.

→ **ASP.NET MVC 2**

This version of ASP.NET MVC was released on March 10, 2010 and targets .NET Framework 3.5 and 4.0. Visual Studio 2008 and 2010 provide support to develop ASP.NET MVC 2 applications. ASP.NET MVC 2 introduced the following key features:

- Strongly typed HTML helpers means
- Data Annotations Attribute
- Client-side validation
- Automatic scaffolding
- Segregating an application into modules
- Asynchronous controllers

→ **ASP.NET MVC 3**

This version of ASP.NET MVC was released on January 13, 2011 and targets .NET Framework 4.0. Visual Studio 2010 provides support to develop ASP.NET MVC 3 applications.

ASP.NET MVC 3 introduced the following key features:

- The Razor view engine
- Improved support for Data Annotations
- Dependency resolver
- Entity Framework Code First support
- ViewBag to pass data from controller to view
- Action filters that can be applied globally

- Unobtrusive JavaScript
- jQuery validation

→ **ASP.NET MVC 4**

This version of ASP.NET MVC was released on August 15, 2012 and targets .NET Framework 4.0 and 4.5. Visual Studio 2010 SP1 and Visual Studio 2012 provide support to develop ASP.NET MVC 4 applications.

ASP.NET MVC 4 introduced the following key features:

- ASP.NET Web API
- Asynchronous Controllers with Task support
- Bundling and minification
- Support for the Windows Azure SDK

→ **ASP.NET MVC 5**

This version of ASP.NET MVC was released on October 17, 2013 and targets .NET Framework 4.5 and 4.5.1. Visual Studio 2013 provides support to develop ASP.NET MVC 5 applications.

ASP.NET MVC 5 introduced the following key features:

- ASP.NET Identity
- ASP.NET Web API2

1.2.5 Architecture of ASP.NET MVC Application

The basic architecture of an ASP.NET MVC application involves the following components:

- The MVC Framework
- The route engine
- The route configuration
- The controller
- The model
- The view engine
- The view

Each of these preceding components communicates to process requests coming to an ASP.NET MVC application. The process of handling an incoming request involves a series of steps that the components of the ASP.NET MVC Framework perform. These steps are as follows:

1. The browser sends a request to an ASP.NET MVC application.
2. The MVC Framework forwards the request to the routing engine.
3. The route engine checks the route configuration of the application for an appropriate controller to handle the request.
4. When a controller is found, it is invoked.
5. When a controller is not found, the route engine indicates that the controller has not been found and the MVC Framework communicates this as an error to the browser.
6. The controller communicates with the model.
7. The controller requests a view engine for a view based on the data of the model.
8. The view engine returns the result to the controller.
9. The controller sends back the result as an HTTP response to the browser.

Figure 1.4 illustrates the preceding steps.

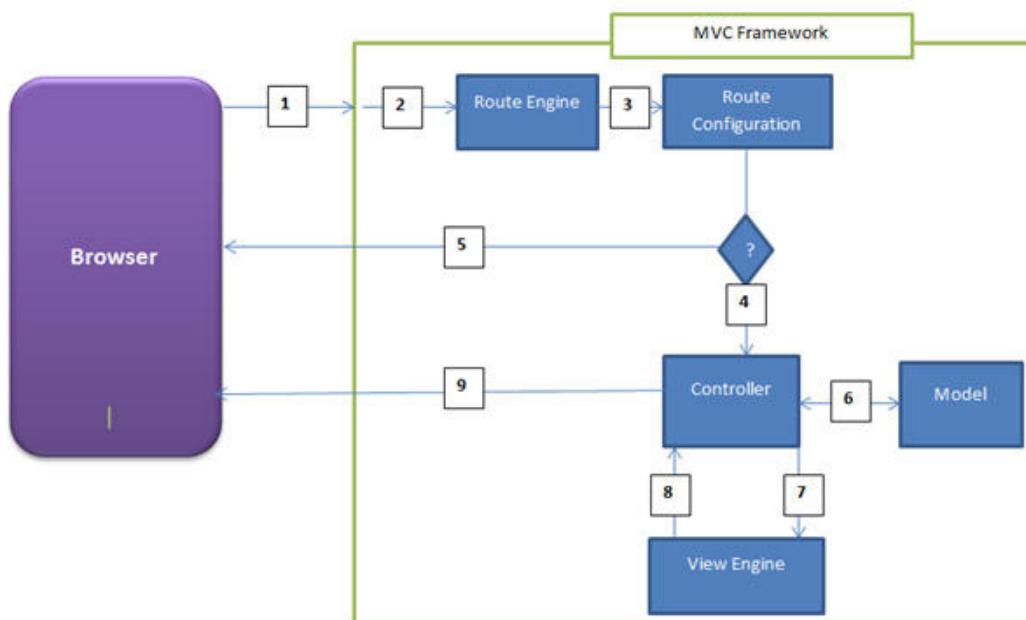


Figure 1.4: Communication in the ASP.NET MVC Architecture

1.3 Supporting Technologies

ASP.NET MVC application supports various technologies to create dynamic and responsive Web application. Some of the supporting technologies that you can use while creating an ASP.NET MVC application are described.

1.3.1 JavaScript

An ASP.NET MVC Web application should be responsive. A responsive Web application is a dynamic and interactive application that enhances user experience. Such responsive Web applications can be created using JavaScript, which is a client-side scripting language that enables a Web application to respond to user requests without interacting with a Web server.

A dynamic and interactive ASP.NET MVC application must implement functionalities, such as an easy to use UI, quick response to the user's request. In addition, it should run in all available browsers.

To achieve this in an ASP.NET MVC application, you can use JavaScript. JavaScript is a client-side scripting language that allows you to develop dynamic and interactive Web applications. When you use JavaScript, your Web applications can respond to user requests without interacting with a Web server. As a result, it reduces the response time to deliver Web pages faster.

1.3.2 JQuery

jQuery is a JavaScript library that simplifies the client-side scripting of HTML. In an ASP.NET MVC application, you will often use jQuery in views to make your views responsive and dynamic. For example, you can use jQuery in a view to create effects, such as fading effects, toggle effects, and animation effects. You can also use jQuery to perform client-side validation of forms.

In addition, jQuery provides several plugins that enable you to enhance the UI elements of a view. For example, you can use jQuery plugins to display calendar controls, accordion controls, tabbed pane controls, and dialog boxes in a view. You can also sort, paginate, and filter tables in views using jQuery plugins.

These plugins are available as the jQuery UI library that is an extension to the base jQuery library.

When you create an ASP.NET MVC project in Visual Studio 2013, the project automatically includes the jQuery libraries within the **Scripts** folder.

Figure 1.5 shows the jQuery libraries in the **Scripts** folder of an ASP.NET MVC project.

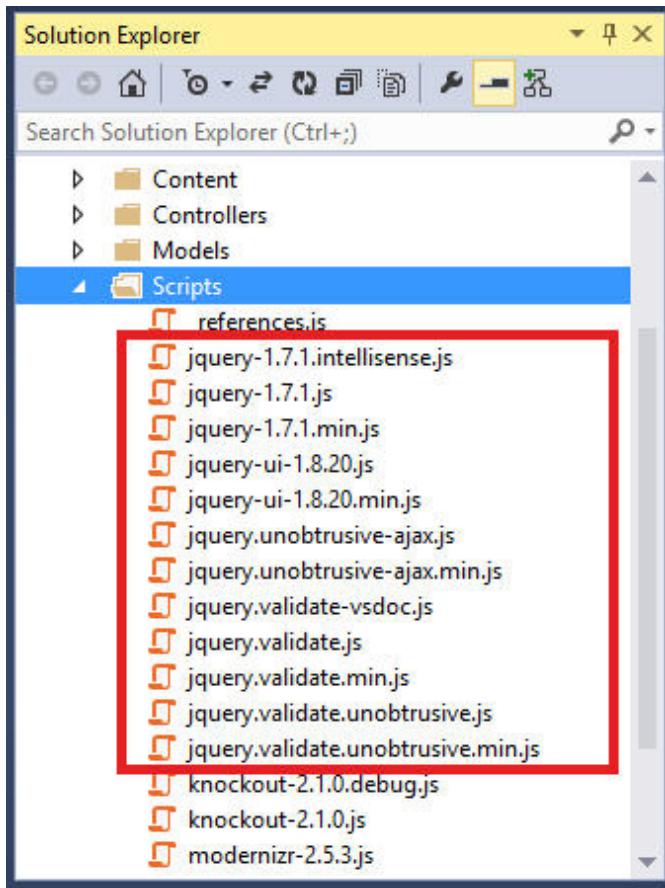


Figure 1.5: jQuery Libraries

To use a jQuery library in a view, you need to refer the library from the view. Code Snippet 1 shows how you can refer a jQuery library from a view.

Code Snippet 1:

```
<script src="/Scripts/jquery-1.7.1.min.js"
type="text/javascript"></script>
<!--JQuery Code -->
</script>
```

1.3.3 Asynchronous JavaScript and XML (AJAX)

AJAX is a Web development technique that is used to create interactive applications. You can use AJAX to asynchronously retrieve data in the background of an ASP.NET MVC application without interfering with the display and behavior of the existing view. AJAX is based on asynchronous communication that provides the ability of an application to send multiple requests and receive responses from the server simultaneously. This enables the users to work continuously on the application without being affected by the responses received from the server.

JavaScript forms an integral part of the AJAX-enabled Web applications because these applications process most of the requests on the client side, unless there is a need to connect to the Web server. The client-side processing of data is handled by using JavaScript object support in AJAX. The XMLHttpRequest object is one of the primary objects used by JavaScript because it enables asynchronous communication between the client and the server.

1.3.4 IIS

An ASP.NET MVC application requires a Web server that enables handling HTTP requests and creates responses. When you request for a Web page on a browser, you type a URL on a browser, for example, `http://mvctest.com/index.html`. This URL consists of the following parts:

- **http**: A protocol to use for exchanging request and response.
- **mvctest.com**: A domain name that maps to an unique IP address.
- **index.html**: A file that you are requesting.

When you submit the URL, the browser first communicates with a name server that converts the domain name, **www.mvctest.com** into an IP address. For example, based on the IP address, the browser creates a connection with a Web server where the IP address is registered. Next, the browser uses the HTTP protocol to send a request to the server, asking for the file. The server, searches for the file and sends back the content of the file as HTML markup to the browser. The browser parses the HTML markup and displays the output to you.

Figure 1.6 shows communication between a browser and the server.



Figure 1.6: Communication between Browser and Server

One such most popularly used Web server is IIS. IIS is a product of Microsoft and is designed to deliver information in high speed and securely. Microsoft has also designed IIS as a platform that developers can target to extend the capabilities of the Internet standards. Therefore, IIS has been designed to be modular. This means, when you install IIS, you can select only those modules that meet your specific functionality requirements.

Some of the modules that you will require to host ASP.NET MVC applications are as follows:

- **Content modules:** Enable controlling how content is delivered to client. For example, this module enables processing requests for files residing on the server. In addition, this module enables displaying a custom error page when a client requested resource is not available.
- **Security modules:** Enable implementing security through various authentication mechanisms, authorization based on URLs, and filtering requests.
- **Compression module:** Enables compressing responses sent to client browser using compression standard, such as Gzip.
- **Caching modules:** Enable caching content and delivering the cached content in subsequent requests for the same resource.
- **Logging and Diagnostics modules:** Enable logging request processing information and response status for diagnostic purposes.

You can deploy and manage ASP.NET MVC applications in IIS using IIS Manager. To access IIS Manager, you need to perform the following steps:

1. Ensure that IIS is installed in your computer.

Note - Windows 8.1 includes (IIS) 8.5 Web server. However, IIS is not installed by default in Windows 8.1.

2. Press the **Windows+R** key combination. The **Run** dialog box appears.
3. Type `inetmgr`. Figure 1.7 shows the **Run** dialog box.

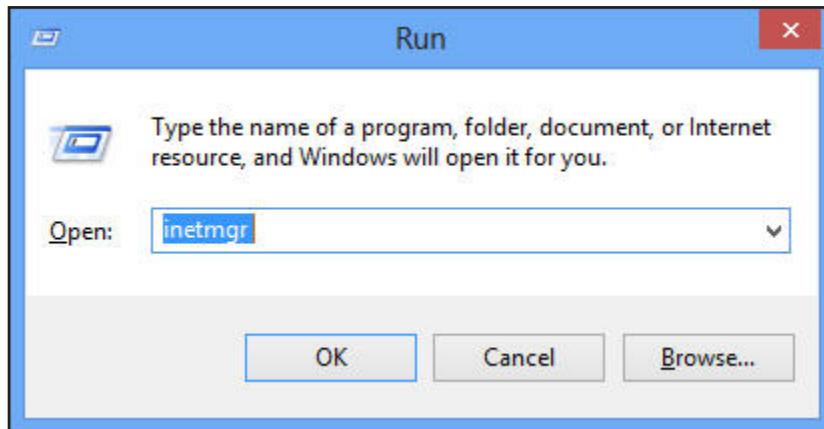


Figure 1.7: Run Dialog Box

4. Click **OK**. The **IIS Manager** window opens.

Figure 1.8 shows the **IIS Manager** window.

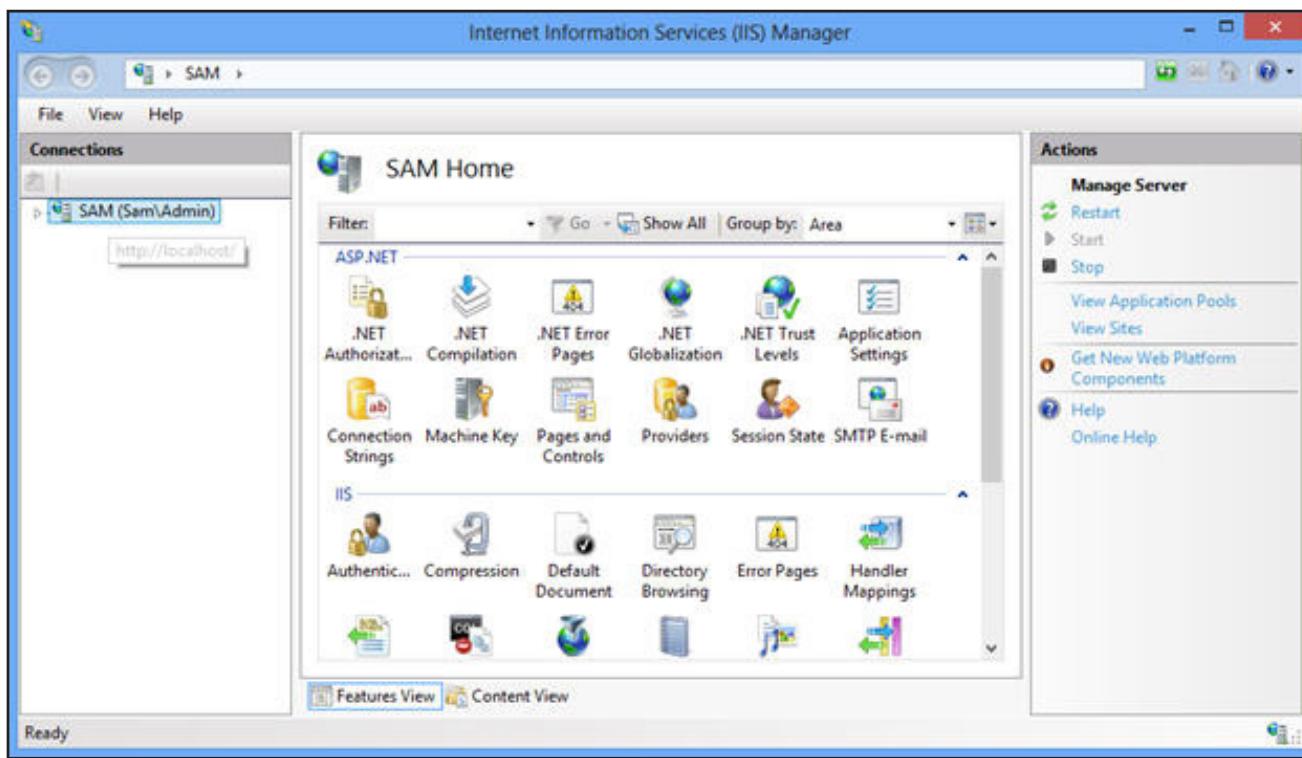


Figure 1.8: IIS Manager

You can use the **IIS Manager** window to configure the features of IIS, such as granting permissions to hosted applications, managing server security, and managing authentication and authorization of applications.

1.3.5 Windows Azure

To understand Windows Azure, you first need to understand about the cloud platform. Prior to the cloud platform, applications were deployed on Internet-accessible servers supported by datacenters. However, individuals and organizations find it difficult to set up such infrastructures. In addition, as applications scale, the problems to maintain such infrastructures become more obvious. As a solution, the cloud platform got introduced where individuals and organizations have the option to host and maintain applications in an infrastructure that they do not have to manage.

One such cloud solution is Windows Azure, provided by Microsoft. As an example, using Windows Azure, you can simply create Word documents, share them, save them without requiring any software to be installed on your computer or upgrading the current hardware configurations of your computer. In enterprise application development, where you are developing enterprise Web applications, you can use Windows Azure as your cloud platform. Windows Azure provides a platform to build applications that can leverage the cloud to meet user needs that can range from a simple task, such as sending an e-mail to managing a complex ASP.NET MVC application that implements an online auction store with global user base.

Windows Azure systems are deployed in Microsoft datacenters. A user instead of downloading, installing, and using a product on their own computers, can use Windows Azure as a service to perform the same functions.

To enable cloud computing Windows Azure provides the following key cloud-based services:

- **Compute services:** Provides the infrastructure to deliver processing power required to run cloud applications through services, such as virtual machines, Web sites, and mobile services.
- **Network services:** Provides the services to deliver cloud applications to users and datacenters.
- **Data services:** Provides the services to enable cloud applications to efficiently store, manage, and secure application data.
- **App services:** Provide the services to enable cloud applications to enhance their performance and security. This service also allows cloud applications hosted on Windows Azure to integrate with other cloud applications.

Figure 1.9 shows the key cloud-based services of Windows Azure.

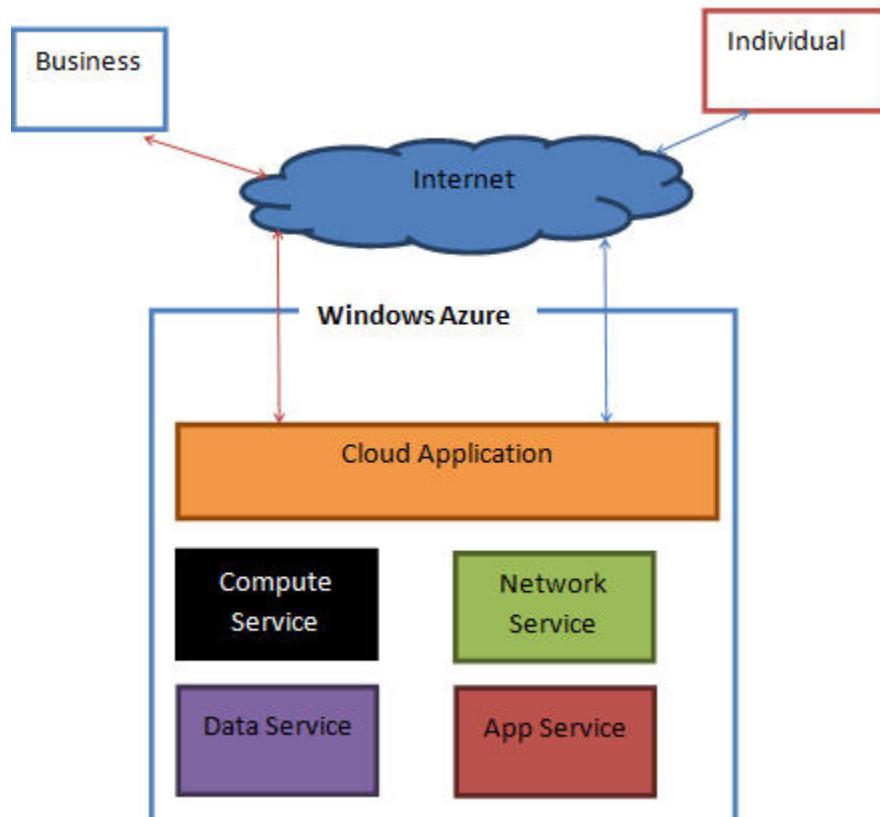


Figure 1.9: Key Cloud-based Services of Windows Azure

1.4 MVC Support in Visual Studio 2013

Visual Studio 2013 simplifies the process of creating ASP.NET MVC applications by providing various in-built templates. Visual Studio 2013 provides an MVC template that automatically creates an MVC application structure with the basic files to run the application. You will now learn how to create an ASP.NET MVC application in Visual Studio 2013.

1.4.1 Creating an ASP.NET MVC Project in Visual Studio 2013

To create an ASP.NET MVC application using Visual Studio 2013, you need to perform the following tasks:

1. Press the **Windows+Q** key.
2. In the **Search** box that appears, start typing ‘Visual Studio 2013’. The **Visual Studio 2013** icon appears under the **Apps** section.
3. Double-click the **Visual Studio 2013** icon. The **Start Page** of Visual Studio 2013 appears.

Figure 1.10 shows the **Start Page** of Visual Studio 2013.

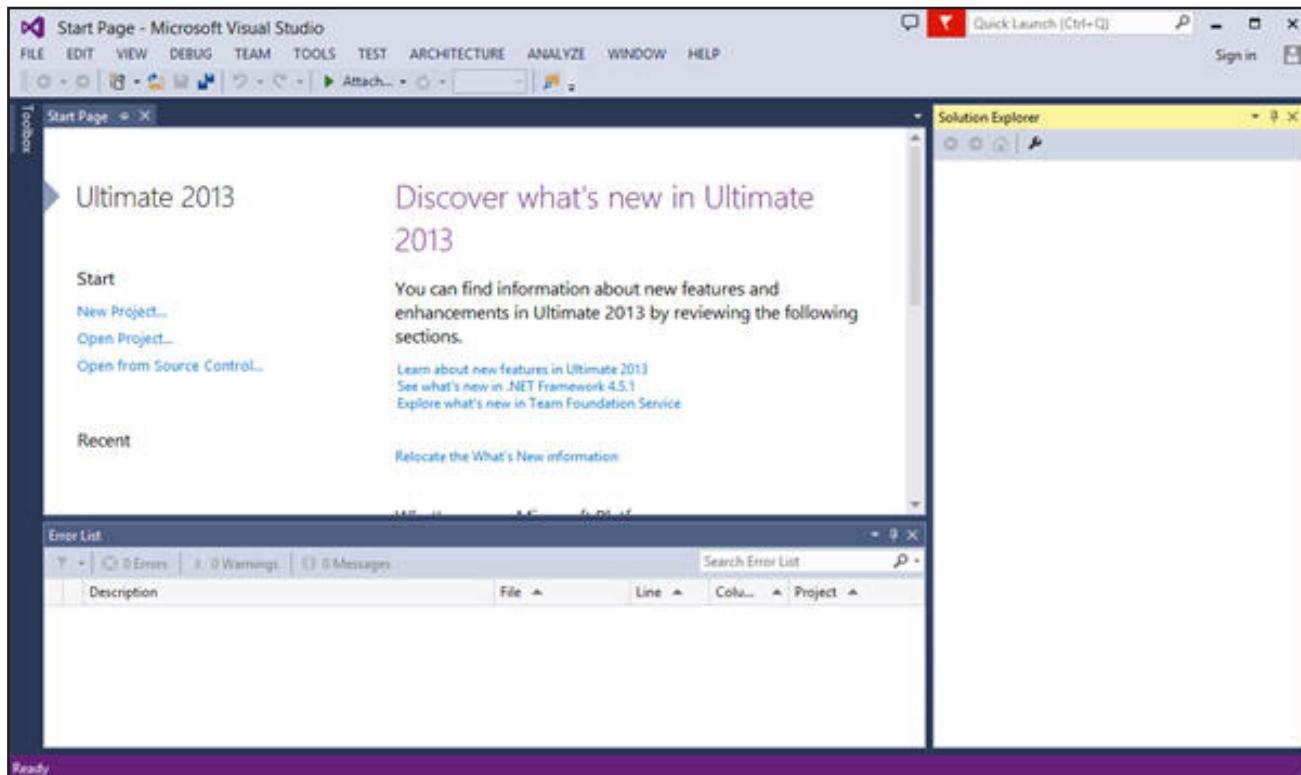


Figure 1.10: Start Page

4. Click **File→New→Projects** menu options in the menu bar of Visual Studio 2013.

Figure 1.11 shows selecting the **File→New→Projects** menu options.

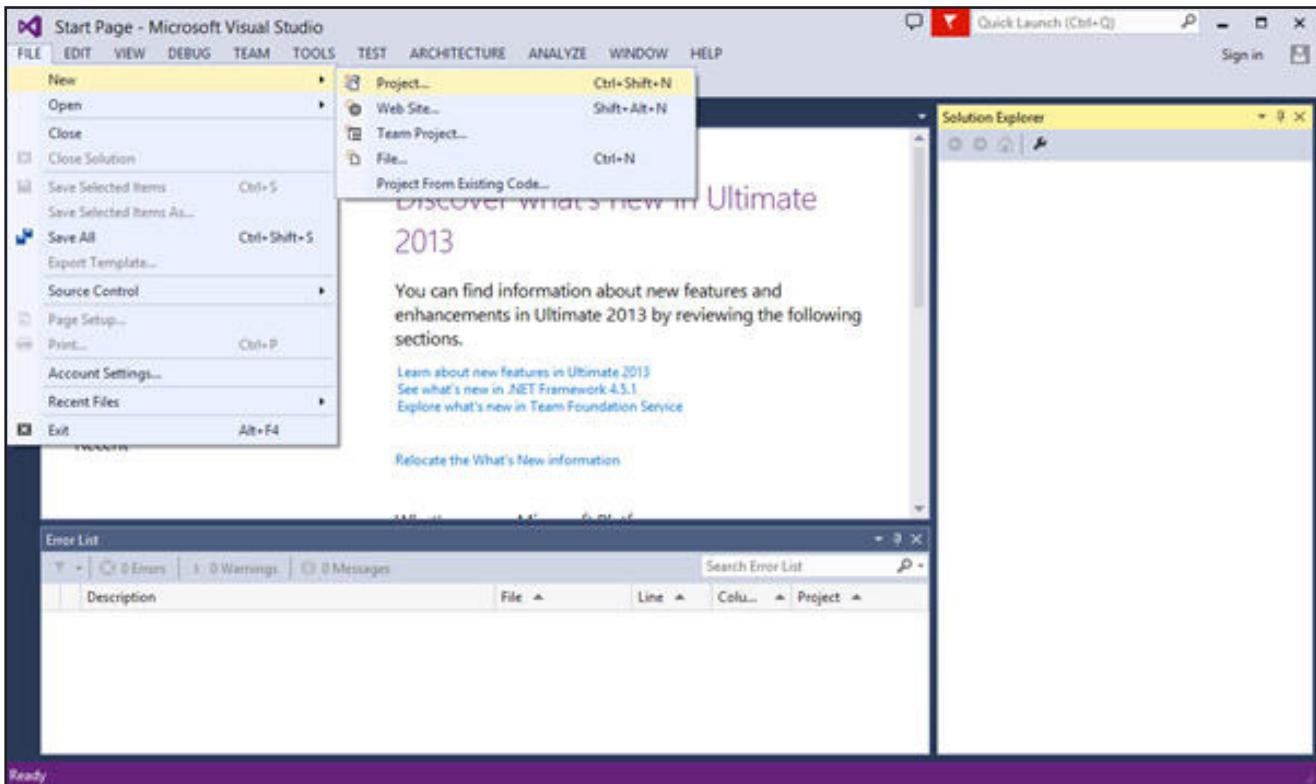


Figure 1.11: Creating New Project

5. In the **New Project** dialog box, select **Web** under the **Installed** section and then, select the **ASP.NET Web Application** template.

Figure 1.12 shows selecting the **Web** and **ASP.NET Web Application** template in the **New Project** dialog box.

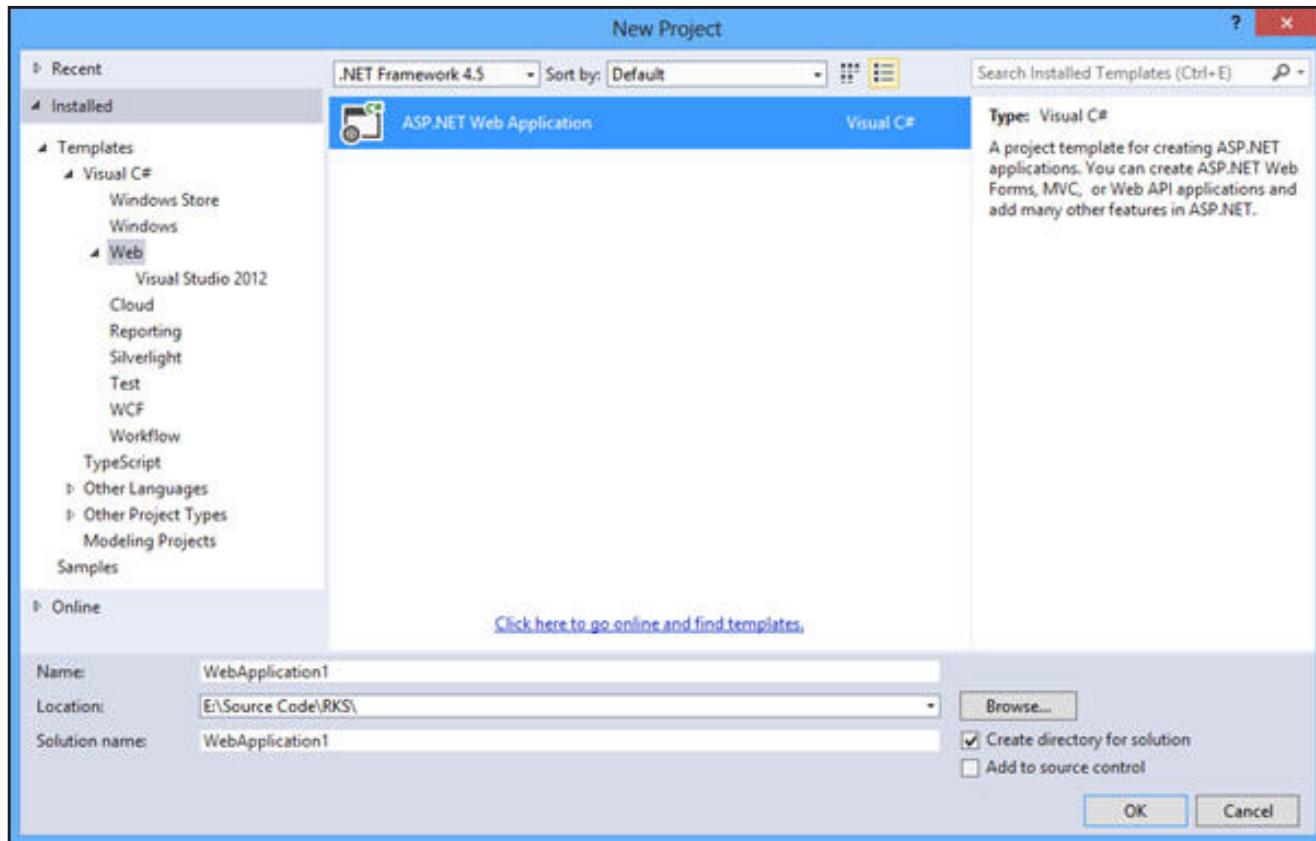


Figure 1.12: New Project Dialog Box

6. Type 'MVC Demo' in the **Name** text field.
7. Click **Browse**. The **Project Location** dialog box appears that allows you to specify the location where the application has to be created.

Figure 1.13 shows specifying the location of the application in the **Project Location** dialog box.

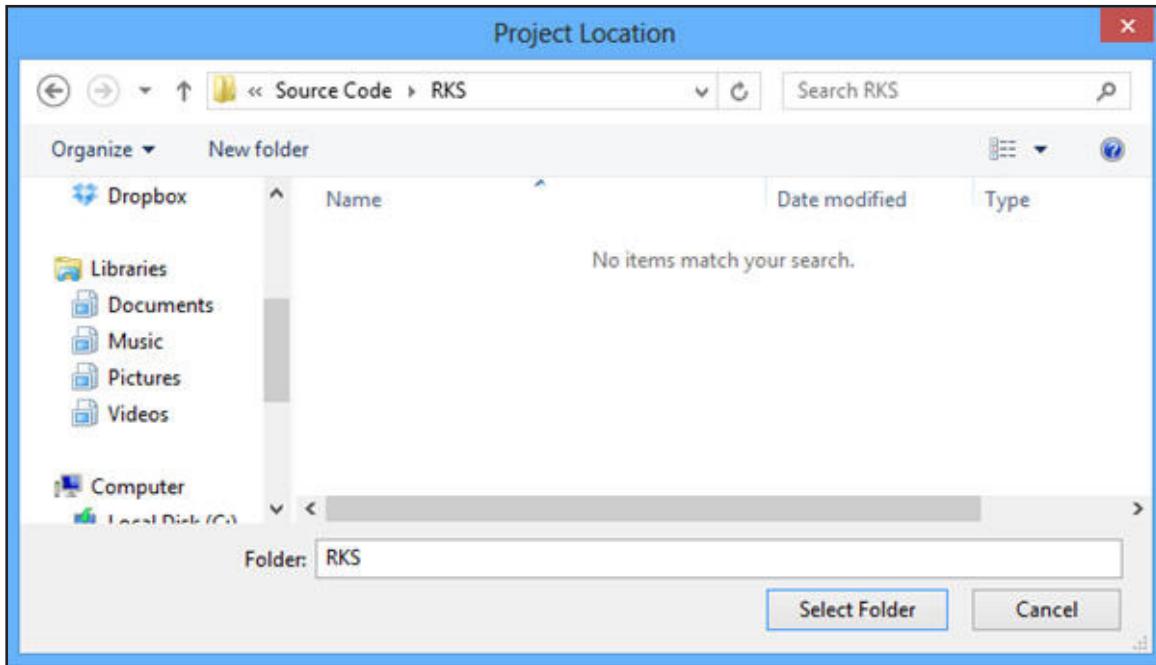


Figure 1.13: Project Location Dialog Box

- Click **Select Folder**. The **New Project** dialog box displays the specified location in the Location field.

Figure 1.14 shows specifying a name and location for the application.

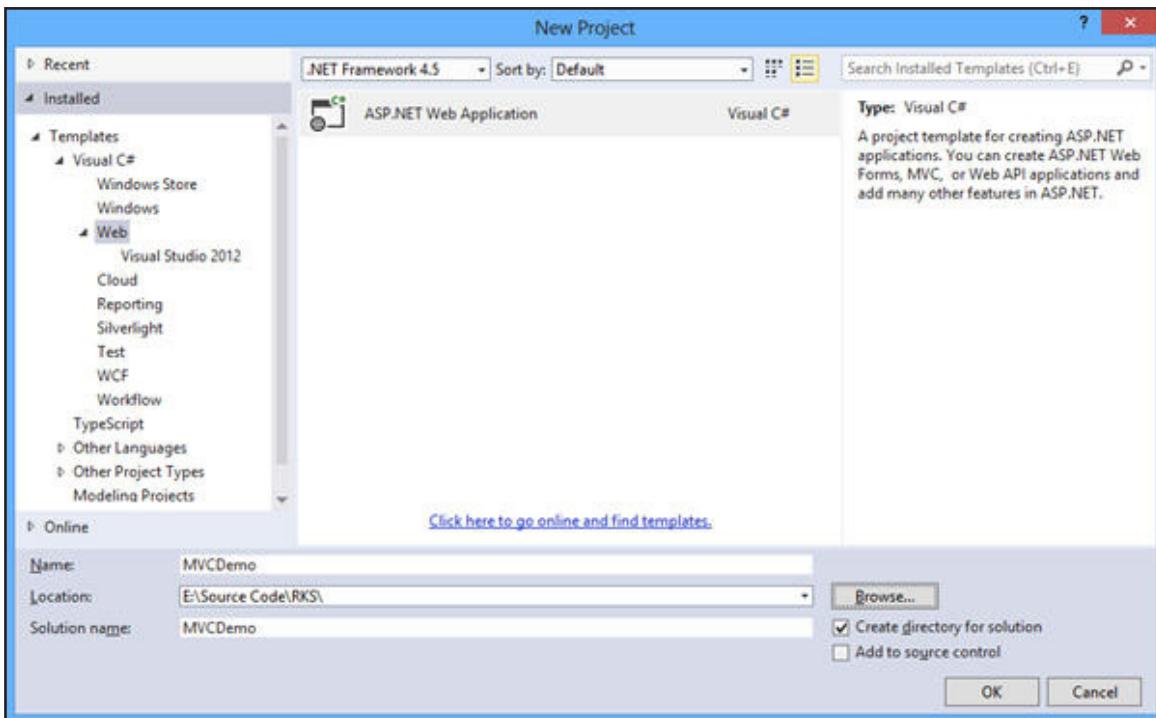


Figure 1.14: Specifying Name and Location

9. Click **OK**. The **New ASP.NET Project – MVCDemo** dialog box appears.
10. Select **MVC** under the Select a template section of the **New ASP.NET Project – MVCDemo** dialog box. Figure 1.15 shows selecting **MVC** under the **Select a template** section.

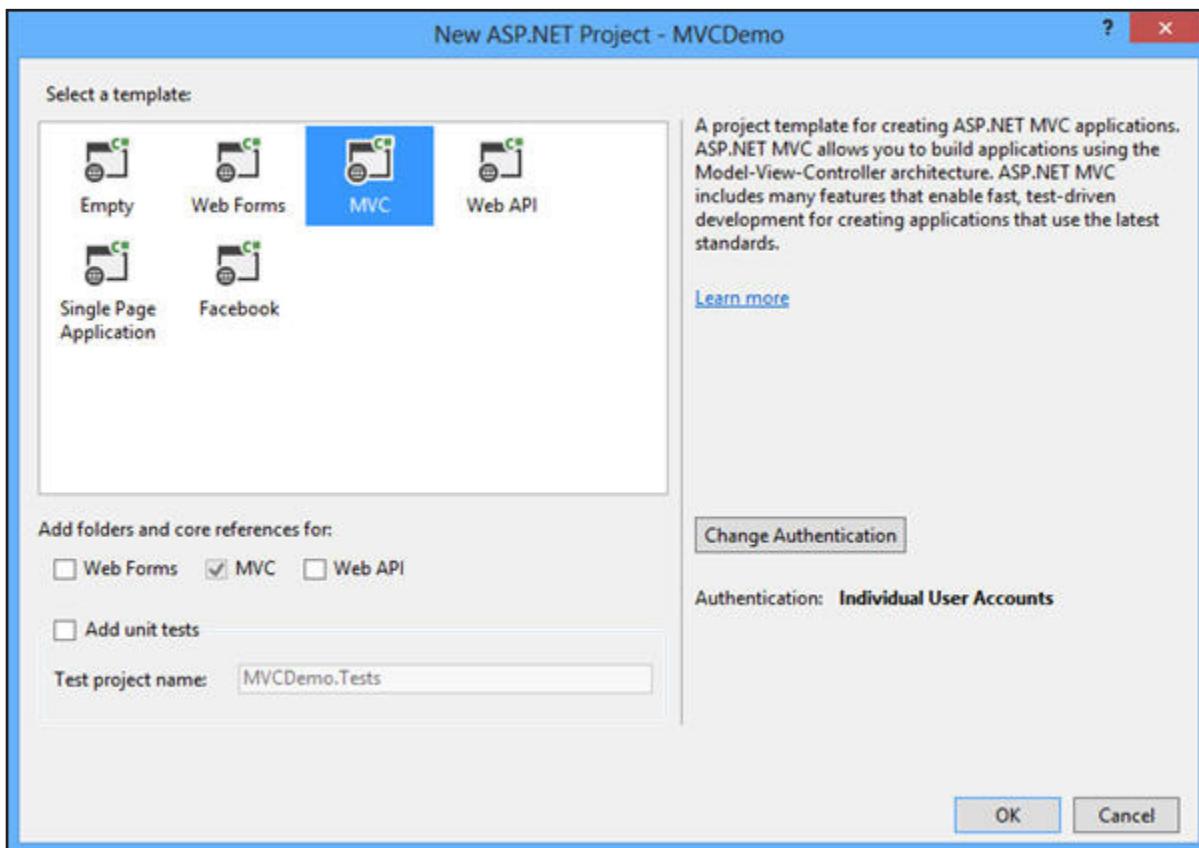


Figure 1.15: Selecting MVC Template

11. Click **OK**. Visual Studio 2013 displays the newly created application.

Figure 1.16 shows the newly created ASP.NET MVC Web application in Visual Studio 2013.

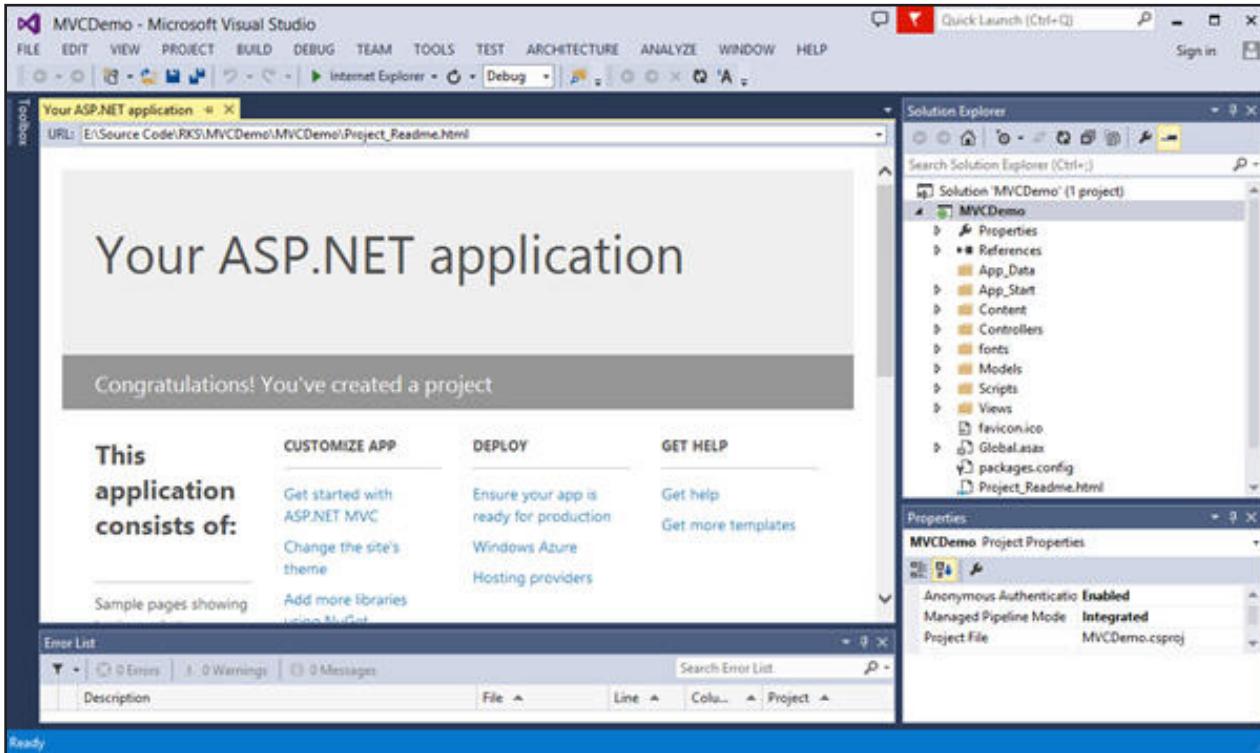


Figure 1.16: Newly Created Application

1.4.2 Structure of an ASP.NET MVC Project

When you use Visual Studio 2013 to create an ASP.NET MVC Web application, it automatically adds several files and folders to the project.

Figure 1.17 shows the files and folders that Visual Studio 2013 creates automatically when you create an ASP.NET MVC application.

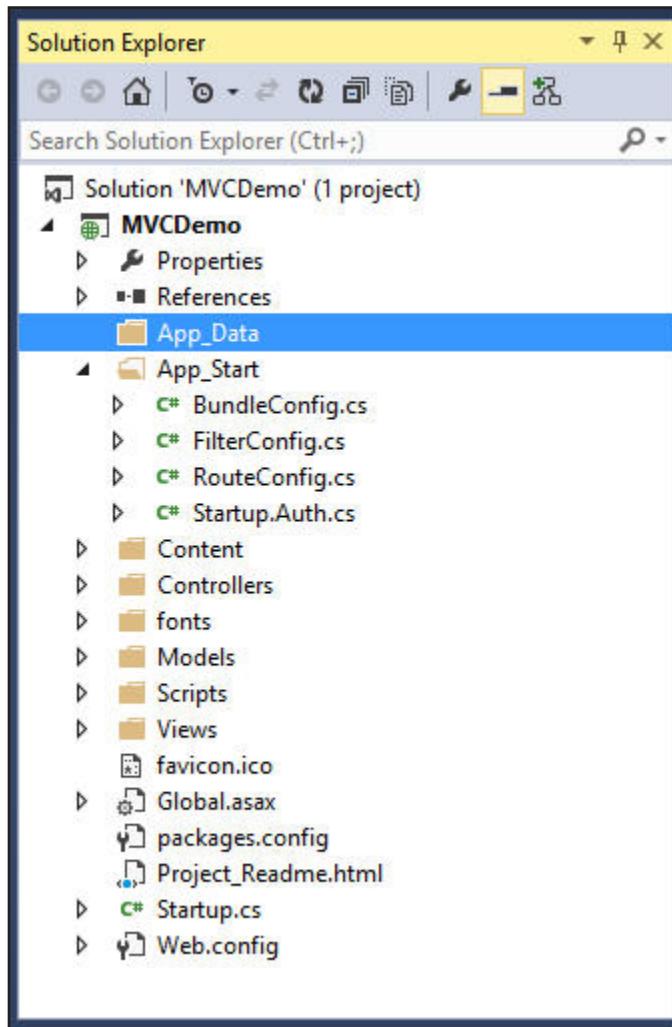


Figure 1.17: Application Directory

The top-level directory structure of an ASP.NET MVC application contains the following folders:

- **Controllers**: Contains the Controller classes that handle URL requests.
- **Models**: Contains the classes that represent and manipulate data and business objects.
- **Views**: Contains the UI template files that are responsible for rendering output, such as HTML.
- **Scripts**: This folder contains the JavaScript library files.
- **Images**: Contains the images that you need to use in your application.

- **Content:** Contains the CSS and other site content, other than scripts and images.
- **Filters:** Contains the filter code.
- **App_Data:** Contains data files that you need to read/write.
- **App_Start:** Contains the files containing configuration code that you can use features such as Routing, Bundling, and Web API.

1.4.3 Executing an ASP.NET MVC Project

To execute an application created in Visual Studio 2013, you need to click **Debug→Start** without debugging from the menu bar. The browser will display the output of the default application. Figure 1.18 shows the output the default application created in Visual Studio 2013.

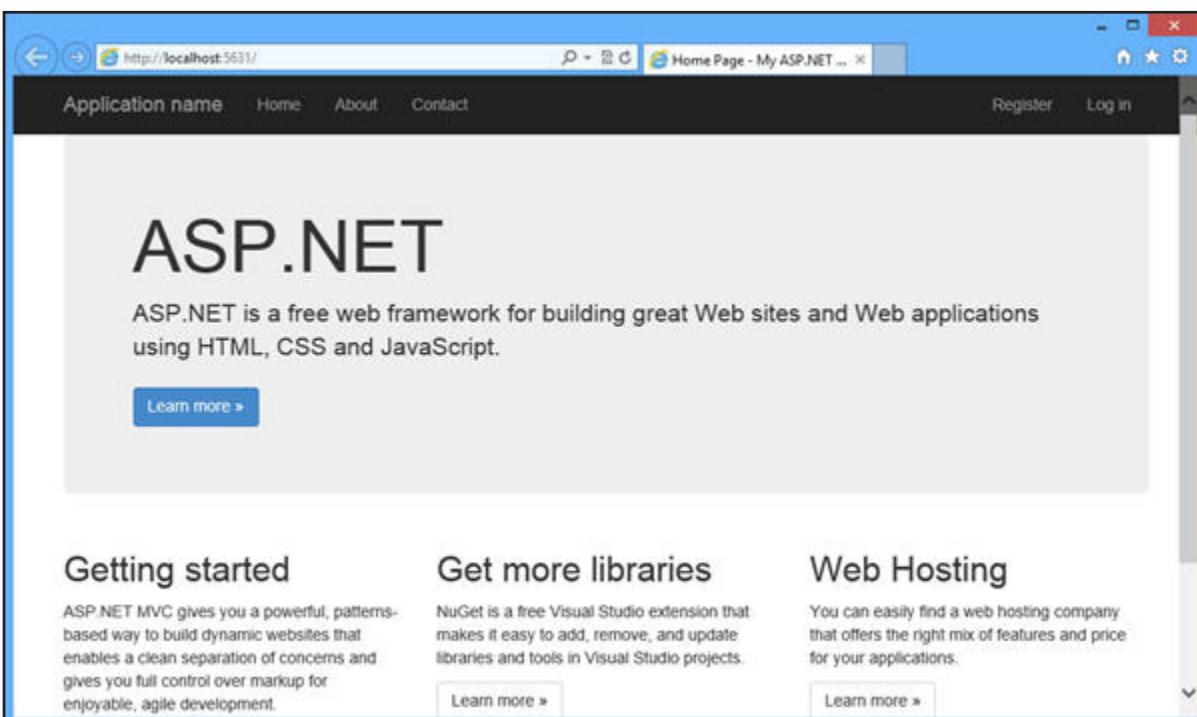


Figure 1.18: Output of the Application

1.5 Check Your Progress

1. Which of the following statements is true regarding the data layer?

(A)	The data layer provides the application data stored in databases to the business logic layer.		
(B)	The data layer allows creating application that enables sharing and accessing information over the Internet that can be accessed globally at any time.		
(C)	The data layer enables users to interact with the application.		
(D)	The data layer controls the flow of execution and communication between the presentation layer and the business layer.		
(E)	The data layer is part of the MVC Framework to validate data that the application sends as response.		

(A)	A	(C)	D
(B)	C	(D)	E

2. Which of the following statements is true regarding the data layer?

(A)	Single-Tier Architecture		
(B)	Two-Tier Architecture		
(C)	Three-Tier Architecture		
(D)	N-Tier Architecture		

(A)	B	(C)	D
(B)	C	(D)	A

3. Which of the following options represents the presentation logic to provide the data of the model?

(A)	Controller		
(B)	Data layer		
(C)	Model		
(D)	View		

(A)	B	(C)	D
(B)	C	(D)	A

4. Which of the following options enables you to develop dynamic and interactive Web applications that can respond to user requests without interacting with a Web server?

(A)	JavaScript
(B)	Controller
(C)	Data layer
(D)	Route engine
(E)	View engine

(A)	B	(C)	D
(B)	C	(D)	A

5. Which of the following folders of an ASP.NET MVC application contains configuration code that you can use features such as Routing, Bundling, and Web API?

(A)	Scripts
(B)	Content
(C)	Filters
(D)	App_Data
(E)	App_Start

(A)	E	(C)	D
(B)	C	(D)	A

1.5.1 Answers

(1)	A
(2)	B
(3)	D
(4)	D
(5)	A



Summary

- ➔ The MVC Framework implements a validation workflow to validate user data.
- ➔ Web applications are programs that are executed on a Web server and accessed from a Web browser.
- ➔ Web applications are typically divided into three layers, where each layer performs different functionalities.
- ➔ The architecture of an application depends on the system in which the layers of the application are distributed and communicated to each other.
- ➔ ASP.NET is a server-side technology that enables you to create dynamic Web applications using advanced features, such as simplicity, security, and scalability, which are based on the .NET Framework.
- ➔ ASP.NET MVC is based on the MVC design pattern that allows you to develop software solutions.
- ➔ An ASP.NET MVC application requires a Web server that enables handling HTTP requests and creates responses.
- ➔ Windows Azure is a cloud solution provided by Microsoft.



To chat with a **Tutor**

Login to www.onlinevarsity.com

Session - 2

Controllers in ASP.NET MVC

Welcome to the Session, **Controllers in ASP.NET MVC**.

In an ASP.NET MVC application, the controller components of the application intercept incoming requests and execute corresponding action methods that implement the functionalities of the application. An ASP.NET MVC application can have multiple controllers and each controller can have one or more action methods. When a request to the application arrives, the MVC Framework routes the request to the appropriate action method of a controller.

The MVC Framework performs the routing using a route engine. The route engine is responsible for mapping the URL of an incoming request to the pre-defined routes of the application. When the route engine finds a matching pre-defined route, the request is routed to the action method of the controller defined by the matching route.

In this Session, you will learn to:

- ➔ Define and describe controllers
- ➔ Describe how to work with action methods
- ➔ Explain how to invoke action methods
- ➔ Explain routing requests
- ➔ Describe URL patterns

2.1 Working with Controllers

In an ASP.NET MVC application, controllers manage the flow of the application. A controller is responsible for intercepting incoming requests and executing the appropriate application code. A controller also communicates with the models of the application and selects the required view to be rendered for the request.

2.1.1 Definition

A controller is a C# class that extends the `Controller` class of the `System.Web.Mvc` namespace. A `Controller` class contains methods, known as action methods. The MVC Framework invoke these methods in response to incoming HTTP requests.

Unlike traditional ASP.NET Web applications, where requests are directly sent to the target page, requests in an ASP.NET Web application are first intercepted by controllers. By using the controllers, the business logic of the application is separated from the presentation logic. This results in clean separation of the work done by programmers and Web designers of an application. In an ASP.NET application, a controller is responsible to:

- Locate the appropriate method to call for an incoming request.
- Validate the data of the incoming request before invoking the requested method.
- Retrieve the request data and passing it to requested method as arguments.
- Handle any exceptions that the requested method throws.
- Help in rendering the view based on the result of the requested method.

2.1.2 Creating a Controller

In ASP.NET MVC, the `ControllerBase` class of the `System.Web.Mvc` namespace is the base class for all controllers. The `Controller` class extends the `ControllerBase` class to provide a default implementation of a controller. To create a controller in an ASP.NET MVC application, you will need to create a C# class that extends the `Controller` class.

Visual Studio 2013 IDE simplifies the process of creating a controller and other components required for an ASP.NET MVC application. Instead of creating a controller manually, you can use Visual Studio 2013 IDE, which also creates the folder structure for the application automatically. In Visual Studio 2013 IDE, you can create a controller by performing the following steps:

1. Right-click the **Controllers** folder in the **Solution Explorer** window.
2. Select **Add→Controller** from the context menu that appears. The **Add Scaffold** dialog box appears.

Figure 2.1 shows the **Add Scaffold** dialog box.

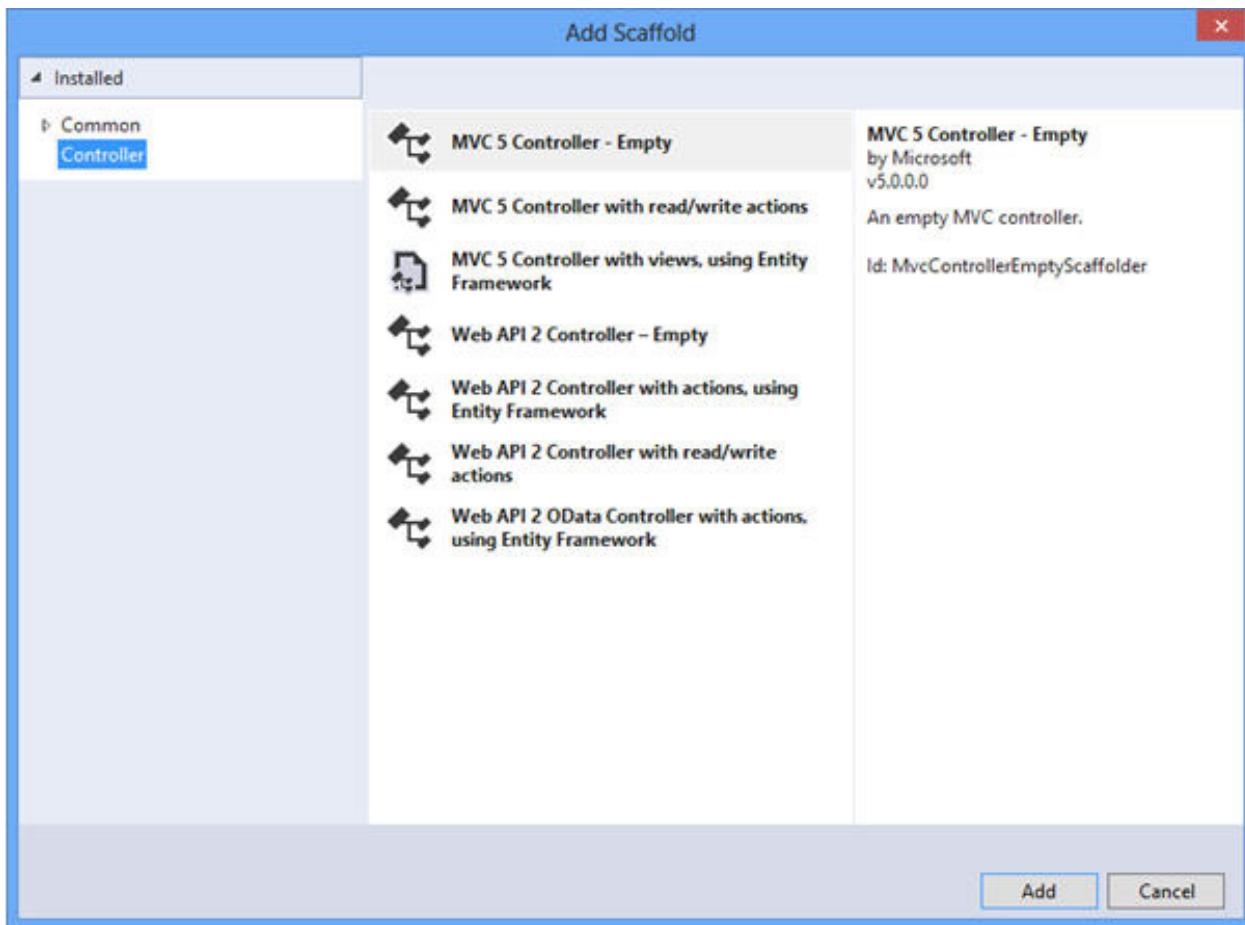


Figure 2.1: Add Scaffold Dialog Box

3. Select the **MVC 5 Controller - Empty** from the **Add Scaffold** dialog box.
4. Click **Add**. The **Add Controller** dialog box appears.
5. Type `TestController` in the **Controller name** text field.

Figure 2.2 shows the **Add Controller** dialog box.

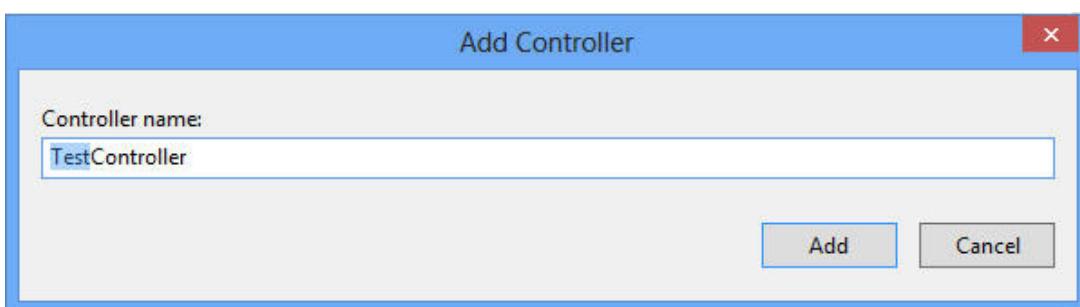


Figure 2.2: Add Controller Dialog Box

6. Click **Add**. The **Solution Explorer** window displays the newly created `TestController` controller under the **Controllers** folder.

Figure 2.3 shows the **Solution Explorer** window that displays the newly created `TestController.cs` under the **Controllers** folder.

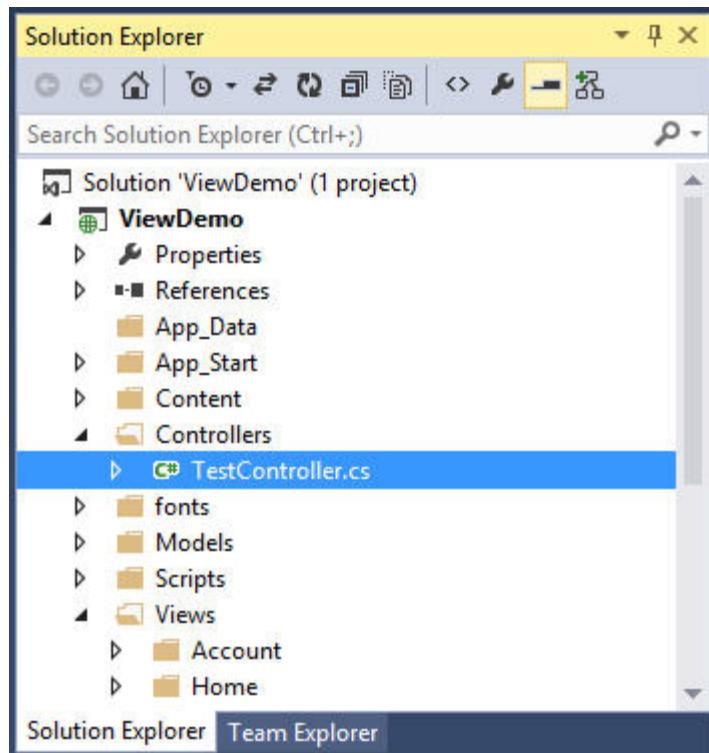


Figure 2.3: TestController.cs

Note - While manually creating a `Controller` class, you should ensure that the class name is suffixed with the word `Controller`.

Following is the syntax for creating a `Controller` class:

Syntax:

```
using System.Web.Mvc;

public class <Controller_Name>Controller:Controller
{
    //Some code
}
```

where,

`Controller_Name`: Is a name of the controller.

Code Snippet 1 shows the skeleton code of a Controller class.

Code Snippet 1:

```
using System.Web.Mvc
public class TestController : Controller
{
    //Some code
}
```

In this code, a controller is created with the name `TestController`.

Note - By convention, you should create a `Controller` class inside the **Controllers** folder of your application directory structure.

2.1.3 Working with Action Methods

A controller class can contain one or more action methods, also known as controller actions. These action methods are responsible for processing the requests that are sent to the controller. An action method typically returns an `ActionResult` object that encapsulates the result of executing the method.

Figure 2.4 shows the working of action methods.

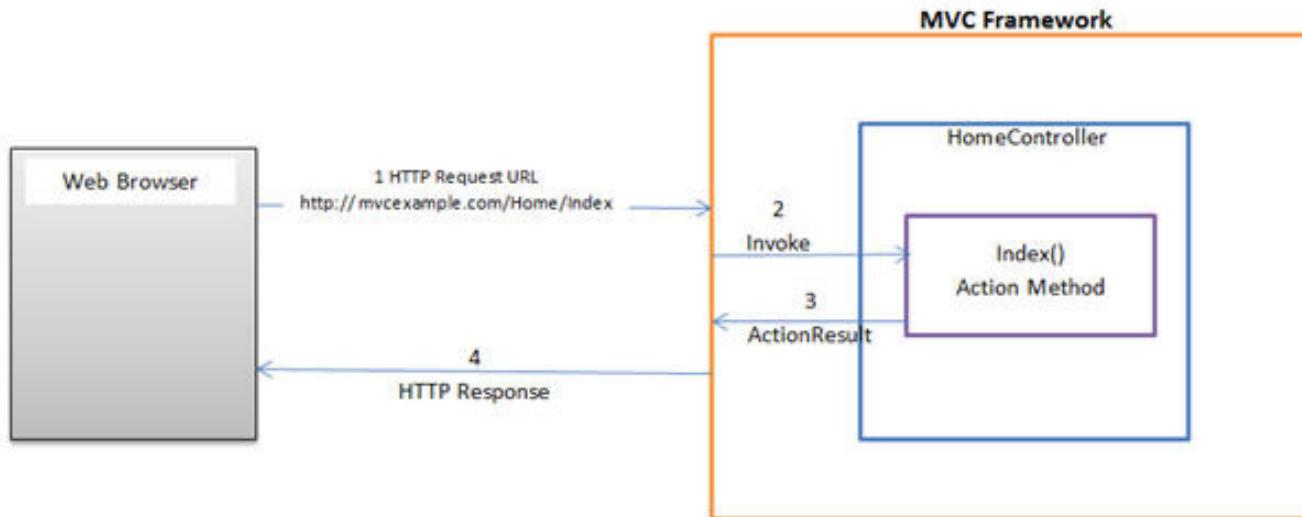


Figure 2.4: Working of Action Methods

The steps in the preceding figure are as follows:

1. The browser sends an HTTP request.
2. The MVC Framework invokes the controller action method, based on the request URL.
3. The action method executes and returns an `ActionResult` object. This object encapsulates the result of the action method execution.

4. The MVC Framework converts an `ActionResult` to HTTP response and sends the response back to the browser.

There are certain rules to create an action method. Some of the rules that you must consider are as follows:

- They must be declared as public.
- They cannot be declared as static.
- They cannot have overloaded versions based on parameters.

The general syntax for creating an action method in a `Controller` class is as follows:

Syntax:

```
public ActionResult <ActionMethod_Name>()
{
    /*Code to execute logic and return the result as
    ActionResult*/
}
```

where,

`<ActionMethod_Name>`: Is a name of the action method.

Code Snippet 2 shows two action methods, with the name `Index` and `About` in the `HomeController` controller class:

Code Snippet 2:

```
using System.Web.Mvc;

public class HomeController : Controller
{
    public ActionResult Index()
    {
        /*Code to execute logic and return the result as
        ActionResult*/
    }

    public ActionResult About()
```

```
{
/*Code to execute logic and return the result as
ActionResult*/
}
}
```

The code creates two action methods, named `Index` and `About` in the `HomeController` controller class. Both these action methods are declared as `public` and return to `ActionResult` objects.

Although, most of the action methods return an `ActionResult` object, an action method can also return other types, such as `String`, `int`, or `bool`.

Code Snippet 3 shows two action methods that return a `bool` and `String` values.

Code Snippet 3:

```
using System.Web.Mvc;

public class HomeController : Controller
{
    public bool IsValid()
    {
        return true;
    }

    public String Contact()
    {
        return "Contact us at www.zynla.com";
    }
}
```

The code creates two action methods, named `IsValid` that returns a `bool` value and `Contact` that returns a `String` value.

2.1.4 Action Results

In an ASP.NET MVC application, the `ActionResult` object that most action methods return, encapsulates the result of that action. `ActionResult` is an abstract base class for all implementing classes that provides different types of results. For example, the `ViewResult` class extends the `ActionResult` class to provide an implementation of an action result that renders a view as an HTML document.

Table 2.1 lists the commonly used classes that extend the `ActionResult` class to provide different implementations of the results of an action method.

Classes	Description
<code>ViewResult</code>	Renders a view as an HTML document.
<code>PartialViewResult</code>	Renders a partial view, which is a sub-view of the main view.
<code>EmptyResult</code>	Returns an empty response.
<code>RedirectResult</code>	Redirects a response to another action method.
<code>JsonResult</code>	Returns the result as JSON, also known as JavaScript Object Notation. JSON is an open standard format to store and exchange text information.
<code>JavaScriptResult</code>	Returns JavaScript that executes on the client browser.
<code>ContentResult</code>	Returns the content based on a defined content type, such as XML.
<code>FileContentResult</code>	Returns the content of a binary file.
<code>FileStreamResult</code>	Returns the content of a file using a <code>Stream</code> object.
<code>FilePathResult</code>	Returns a file as a response.

Table 2.1: Classes

2.1.5 Invoking Action Methods

In an ASP.NET MVC application, you can create multiple action methods in a controller. By default, you can invoke an action method by specifying a URL in the Web browser containing the name of the controller and the action method to invoke.

The general syntax to invoke an action method is as follows:

Syntax:

`http:// <domain_name> /<controller_name>/<actionmethod_name>`

where,

`<domain_name>`: Is the domain name of the application.

`<controller_name>`: Is the name of the controller without the `Controller` suffix.

`<actionmethod_name>`: Is the name of the action method to invoke.

As an example, if you need to invoke a `Registration` action method in the `HomeController` controller of an application with the domain name, **mvceexample.com**, you need to specify the following URL in the Web browser.

`http:// mvceexample.com/Home/Registration`

When the preceding URL is sent to the application through a Web browser, the MVC Framework performs the following tasks:

1. Searches for the `HomeController` controller class.
2. Searches for the `Registration()` action method in the `HomeController` controller class.
3. Executes the `Registration()` action method.
4. Returns the response back to the browser.

Figure 2.5 illustrates the preceding steps.

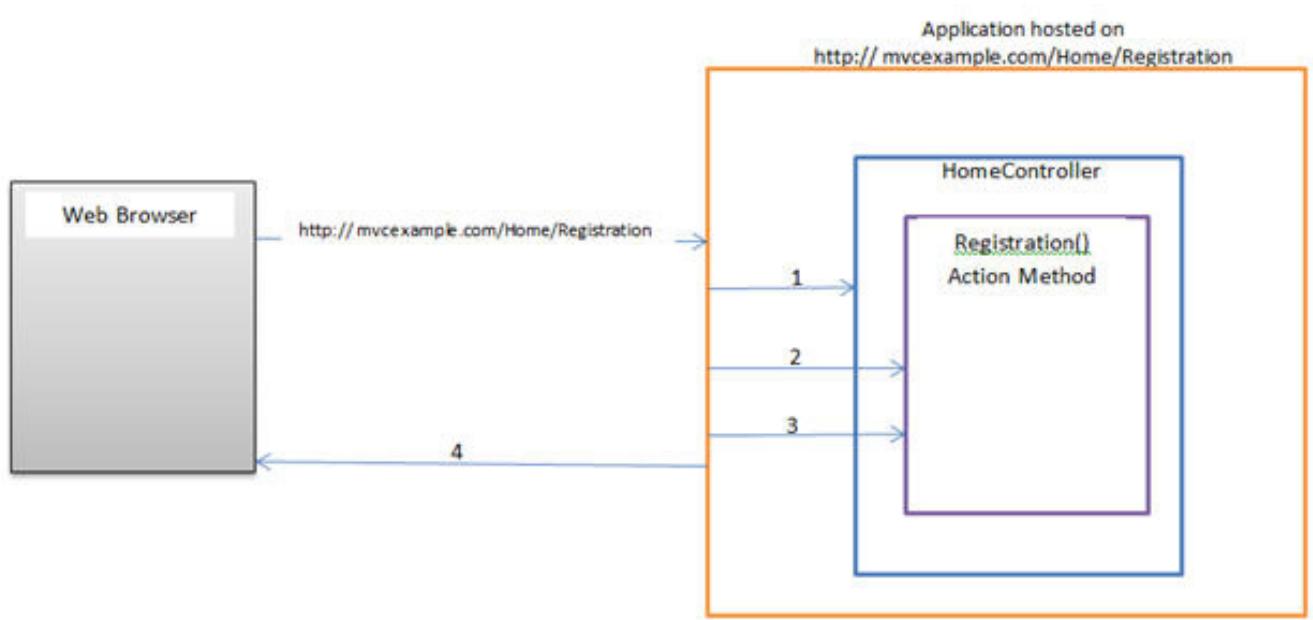


Figure 2.5: MVC Application Processing URL Request

2.1.6 Passing Parameters

Sometimes, you may need to provide input other than the Web page name, while requesting for a Web page. As an example, you may need to provide a **student ID** as the input to view details of a particular student. You can send the student ID to the server as a query string in the URL.

Note - A query string is a part of a URL that is used to pass data from a browser to a Web application. The data is passed as a **name value pair**. The part of a URL that follows the ? character is a query string.

For example, in the following URL,

`Id=007` is a query string.

`http://www.mvceexample.com/home/detail?Id=007`

For example, consider the following URL:

`http://www.mvceexample.com/student/details?Id=006`

In an MVC application, the preceding URL will invoke the `Details` action method of the `StudentController` controller class. The URL also contains an `Id` parameter with the value `006`.

The `Details` action method must accept an `Id` parameter of type `string` in order to return student records based on the `Id` value.

Note - The action method must have a parameter whose name is the same as the name of parameter passed in the query string.

Code Snippet 4 shows the `Details` action that accepts an `Id` parameter.

Code Snippet 4:

```
public ActionResult Details(string Id)
{
    /*Return student records based on the Id parameter as an
ActionResult object*/
}
```

2.2 Routing Requests

In traditional ASP.NET application that does not have any routing feature, the incoming request for a URL is mapped to a physical file, such as an `.aspx` file that handles the request. However, MVC Framework introduces routing that allows you to define URL patterns with placeholders that maps to request URLs pattern.

In an ASP.NET MVC application, routing defines how the application will process and respond to incoming HTTP request. You can use URLs that properly describes the controller action to which the request needs to be routed.

2.2.1 Uses of Routing

Routing is a process that maps incoming requests to specified controller actions. The two main functions of routing are as follows:

- Mapping incoming requests to controller action.
- Constructing outgoing URLs corresponding to controller actions.

Routing is achieved by configuring route patterns in the application. The configuration process involves the following tasks:

- Creating the route patterns
- Registering the patterns with the route table of the MVC Framework

At the time of creating an ASP.NET MVC application, the application can register multiple routing patterns with the MVC Framework's route table. This provides the information on how the routing engine process requests that matches those patterns. On receiving a request, the routing engine starts matching the requested URL with the registered URL patterns. Once the request URL matches with a pattern, the routing engine route the request to the corresponding action and the matching process stops.

2.2.2 The Default Route

An MVC application requires a route to handle user requests. When you create an ASP.NET MVC application in Visual Studio. NET, a route is automatically configured in the `RouteConfig.cs` file. The `RouteConfig` class contains a `MapRoute()` method.

Code Snippet 5 shows the `MapRoute()` method.

Code Snippet 5:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

In this code, `routes`, which is of type `System.Web.Routing.RouteCollection` represents a collection of routes for the application. The `MapRoute()` method defines a route named, `Default`, a URL pattern, and a default route. The default route is used if the request URL does not match with the defined URL pattern defined in the `MapRoute()` method. For example, if a request URL does not contain the name of a controller and an action, the request will be routed to the `Index` action of the `Home` controller.

2.2.3 Registering the Default Route

An MVC application contains a `Global.asax` file which initializes the application with the features of the MVC Framework when the application starts. The `Global.asax` file contains an `MVCApplication` class with the `Application_Start()` method. When the application starts, the MVC Framework invokes the `Application_Start()` method. In this method, you need to register the default route so that the framework uses the route when a request starts coming to the application.

Code Snippet 6 shows the `MVCApplication` class of the `Global.asax` file.

Code Snippet 6:

```
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
namespaceUrlsAndRoutes {
    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            /*Code for registering other MVC components*/
        }
    }
}
```

In this code, the `MVCApplication` class extends the `System.Web.HttpApplication` class that defines the common features required by the application objects in an ASP.NET application. The `Application_Start()` method calls the `RouteConfig.RegisterRoutes()` method to register the default route to be used in the application.

2.2.4 URL Patterns

When you create a route, you need to define a URL pattern for the route. The route engine of the MVC Framework maps the URL of a request with the URL pattern. A URL pattern contain literal values and placeholders separated by the slash (/) character. The placeholders in a URL pattern is enclosed in braces, { and }.

For example, consider the following URL pattern:

`{controller}/{action}/{id}`

The preceding URL pattern consists of three placeholders `{controller}`, `{action}`, and `{id}`.

A URL that will match the preceding pattern is as follows:

`http://www.mvceexample.com/student/records/36`

When the routing engine matches the preceding URL with the URL pattern, it performs the following actions:

- ➔ Assign `student` as the value of the `{controller}` placeholder.
- ➔ Assigns `records` as the value of the `{action}` placeholder.
- ➔ Assign `36` as the value of the `{id}` placeholder.

A URL parameter can also have a combination of literal values and placeholders as shown in the following code snippet:

```
student/{action}/{id}
```

Some examples of URLs that will match with the preceding URL pattern are as follows:

- ➔ `http://www.mvceexample.com/student/records/36`
- ➔ `http://www.mvceexample.com/student/delete/21`
- ➔ `http://www.mvceexample.com/student/view/23`

2.2.5 Ordering Routes

Occasionally, you may need to register multiple routes in an ASP.NET MVC application. In such situation, you need to configure the sequence in which the routes will execute. This is because the route engine starts matching a request URL with a URL pattern starting from the first registered route. When a matching route is encountered, the route engine stops the matching process. This may cause unexpected results.

Code Snippet 7 demonstrates the following two routes.

Code Snippet 7:

```
routes.MapRoute(
    name: "general",
    url: "{controller}/{action}",
    defaults: new { controller = "Home", action = "Index" });
routes.MapRoute(
    name: "manager",
    url: "Manager/{action}",
    defaults: new { controller = "Manager", action = "Browse" })
;
```

This code contains two placeholders and sets the default value of the controller parameter to `Home` and the action parameter to `Index`. On the other hand, the second route contains a literal, `Manager`, and a placeholder, and sets the default value of the controller parameter to `Manager` and the action parameter to `Browse`.

Now, consider the following URL:

`http://www.mvceexample.com/Manager`

Ideally, the preceding URL should invoke the `Browse` action method of the `Manager` controller. However, the route engine will check the first route, find a match, and will route the request to the `Index()` action method of the `Manager` controller. The route engine will then, stop any further route matching. However, the `Manager` controller might not have an `Index()` method and as a result, an exception will be thrown.

To prevent such problems, you need to configure the route sequence so that the routing engine checks the route named, `manager` ahead of the route named, `general`.

Code Snippet 8 shows the new route sequence.

Code Snippet 8:

```
routes.MapRoute(
    name: "manager",
    url: "Manager/{action}",
    defaults: new { controller = "Manager", action = "Browse" }
);

routes.MapRoute(
    name: "general",
    url: "{controller}/{action}",
    defaults: new { controller = "Home", action = "Index" }
);
```

In this code, the route named, `manager` is declared ahead of the route named, `general`. As a result, the routing engine will route the following URL to the `Browse` action method of the `Manager` controller.

`http://www.mvceexample.com/Manager`

2.2.6 Constraining Route

In an ASP.NET MVC application, the routing engine enables you to apply constraints around the placeholder values. You can use constraints in scenarios where an application have identical route URLs, but based on the application requirement, the route engine should resolve the URLs to different controllers or actions. For example, you may need to place extra validation constraints to ensure that the route matches only with the requests that meet the validation constraints.

Code Snippet 9 shows a route with a route constraint.

Code Snippet 9:

```
routes.MapRoute(
    "Product",
    "{controller}/{action}/{id}",
    new { controller = "Product", action = "Browse", id = UrlParameter.Optional },
    new { id = "(Jewellery|Jeans|Mobile)" }
);
```

In this code, a constraint is applied to the route, so that `id` placeholder can have only one of the Jewellery, Jeans, and Mobile values.

Table 2.2 describes whether the routing engine will match different URLs based on the routing constraints.

URL	Matching Results
<code>http://www.mvceexample.com</code>	Yes. The default values of the controller and action are specified as Product and Browse respectively.
<code>http://www.mvceexample.com/Product</code>	Yes. The default value of the action is specified as Browse.
<code>http://www.mvceexample.com/Product/Browse</code>	Yes. The id specified as an optional parameter.
<code>http://www.mvceexample.com/Product/Browse/Jewellery</code>	Yes. Jewellery specified in the URL is present in the list containing valid values for id parameter.
<code>http://www.mvceexample.com/Product/Browse/Jeans</code>	Yes. Jeans specified in the URL is present in the list containing valid values for id parameter.
<code>http://www.mvceexample.com/Product/Browse/Mobile</code>	Yes. Mobile specified in the URL is present in the list containing valid values for id parameter.
<code>http://www.mvceexample.com/Product/Browse/Laptop</code>	No. Laptop specified in the URL is not present in the list containing valid values for id parameter.
<code>http://www.mvceexample.com/Product/Browse/Glasses</code>	No. Glasses specified in the URL is not present in the list containing valid values for id parameter.

Table 2.2: URLs with Matching Results

Consider another example, you need to create a route with a pattern where the `id` placeholder can have only numeric values. You can create a route constraint using regular expression to ensure that the `id` parameter of the URL can contain only an integer value.

Code Snippet 10 shows a route with a route constraint which specifies that the `id` placeholder can match only with an `integer` value.

Code Snippet 10:

```
routes.MapRoute(
    name: "Product",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Product", action = "Browse", id = UrlParameter.Optional },
    constraints: new { id = @"\d*" }
);
```

This code shows a route with three placeholders, `controller`, `action`, and `id`. In this route, the default value for the `controller` placeholder is set to `Product`, the `action` placeholder is set to `Browse`. In addition, a constraint is applied to an `id` optional parameter. This constraint uses a regular expression to specify that the `id` parameter can match only with an `integer` value.

2.2.7 Ignoring a Route

The MVC Framework provides you the flexibility to ignore routes. For example, consider that, an ASP.NET MVC application has files with `.axd` extension that are used for common ASP.NET handlers, such as `Trace.axd` and `WebResource.axd`. In the application, requests for such `.axd` file should be handled as normal requests to ASP.NET and not by the routing engine. In such situations, you can specify a route that the MVC Framework should ignore for requests to `.axd` files.

The `IgnoreRoute()` method of the `RoutesTable` class enables you to specify a route that the MVC routing engine should ignore.

Code Snippet 11 shows how to ignore a route.

Code Snippet 11:

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

In this code, the `routes.IgnoreRoute()` method specifies that resources with the `.axd` extension should be ignored by the route engine and served directly as response.

Note - When you ignore routes in an application, you should call the `IgnoreRoute()` method before the standard routes.

2.3 Check Your Progress

1. Which of these statements about controller are true?

(A)	Controller manages the flow of the application in ASP.NET MVC application.		
(B)	Controller allows you to define URL patterns with placeholders that maps to request URLs pattern.		
(C)	Controller automatically configures a route in the <code>RouteConfig.cs</code> file when you create an ASP.NET MVC application.		
(D)	Controller communicates with the models of the application and selects the required view to be rendered for the request.		
(E)	Controller allows you to create a route constraint using regular expression.		

(A)	A, B	(C)	A, D
(B)	C, D	(D)	B, D

2. Match the descriptions against their corresponding `ActionResult` implementation classes.

Classes		Description	
(A)	EmptyResult	(1)	Returns the content of a binary file.
(B)	RedirectResult	(2)	Renders a view as an HTML document.
(C)	FileContentResult	(3)	Returns a file as a response.
(D)	FilePathResult	(4)	Redirects a response to another action method.
(E)	ViewResult	(5)	Returns an empty response.

(A)	A-1, B-4, C-3, D-5, E-2	(C)	A-3, B-4, C-5, D-1, E-2
(B)	A-2, B-1, C-3, D-4, E-5	(D)	A-5, B-4, C-1, D-3, E-2

3. Which of the following methods does the MVC Framework invokes when an application starts?

(A)	Index ()
(B)	Application_Start ()
(C)	MapRoute ()

(D)	RegisterRoute()
(E)	IgnoreRoute()

(A)	B	(C)	D
(B)	C	(D)	A

4. Consider the following code snippet that applies a route constraint in an ASP.NET MVC application:

```
routes.MapRoute(
    "News",
    "{controller}/{action}/{id}",
    new { controller = "News", action = "Browse", id = UrlParameter.Optional },
    new { id = "(Sports|Politics|Technology)" }
);
```

Assuming the application is hosted on <http://www.mvceexample.com>, which of the following URLs will match with the preceding route constraint?

(A)	http://www.mvceexample.com
(B)	http://www.mvceexample.com/Browse
(C)	http://www.mvceexample.com/News
(D)	http://www.mvceexample.com/News/Browse/Technology
(E)	http://www.mvceexample.com/News/Browse/Local

(A)	A, C, D	(C)	D, E
(B)	C, E	(D)	A, B, E

5. Which of the following statements are false about routing?

(A)	Routing is a process that maps incoming requests to specified controller actions.
(B)	Routing provides input other than the Web page name while requesting for Web pages.
(C)	Routing allows you to create multiple action methods in a controller.
(D)	Routing allows rendering a partial view, which is a sub-view of the main view.
(E)	Routing is achieved by configuring route patterns in an application.

(A)	A, B, C	(C)	B, C, D
(B)	C, D, E	(D)	B, D, E

2.3.1 Answers

(1)	C
(2)	D
(3)	A
(4)	A
(5)	C



Summary

- A controller is responsible for intercepting incoming requests and executing the appropriate application code.
- To create a controller in an ASP.NET MVC application, you will need to create a C# class that extends the Controller class.
- A controller class can contains one or more action methods, also known as controller actions.
- Although, most action methods return an ActionResult object, an action method can also return other types, such as String, int, or bool.
- Routing is a process that maps incoming requests to specified controller actions.
- When you create a route, you need to define a URL pattern that can contain literal values and placeholders separated by the slash (/) character for the route.
- The routing engine allows you to apply constraints around the placeholder values and also ignore routes.

Technowise



Are you a
TECHNO GEEK
looking for updates?

Login to

www.onlinevarsity.com

Session - 3

Views in ASP.NET MVC

Welcome to the Session, **Views in ASP.NET MVC**.

In an ASP.NET MVC application, the view components display the UI of the application. Views are used to display both static and dynamic content. Views are also used to display form fields to accept user inputs. The controllers of an application can pass data to a view using the `ViewData`, `ViewBag`, and `TempData` objects. The MVC Framework provides view engines that generate Hypertext Markup Language (HTML) from views. The two built-in view engines of the MVC Framework are, Web Form View Engine and Razor View Engine. To create views for the Razor view engine, you need to use special syntax, known as Razor syntax. In addition to view engines, the MVC Framework provides a set of HTML helper methods that generates HTML code in a view for you.

In this Session, you will learn to:

- ➔ Define and describe views
- ➔ Explain and describe the Razor view engine
- ➔ Define and describe the HTML helper methods

3.1 Working with Views

Most controller actions in an ASP.NET MVC application require displaying dynamic content, typically in HTML format. To display these HTML content to the user, you can instruct the controller action in the application to return a view. A view in an ASP.NET MVC application provides the UI of the application to the user. Views are used to display content of an application and also to accept user inputs.

Typically, a view uses model data to create this UI. An example would be an edit view of a Products table that displays UI fields, such as text boxes, drop-down lists, and check boxes to edit the current state of a Product model.

A view contains both HTML markup and code that runs on the Web server. While creating a view, you should specify the view engine that will process the server-side code.

3.1.1 View Engines

View engines are part of the MVC Framework that converts the code of a view into HTML markup that a browser can understand. The MVC Framework provides two view engines:

- **Web form view engine:** Is a legacy view engine for views that uses the same syntax as ASP.NET Web Forms. This view engine was the default view engine for MVC 1 and MVC 2.
- **Razor view engine:** Is the default view engine starting from MVC 3. This view engine does not introduce a new programming language, but instead introduces new markup syntax to make transitions between HTML markups and programming code simpler.

3.1.2 Specifying the View for an Action

While creating an ASP.NET MVC application, you often need to specify a view that should render the output for a specific action. When you create a new project in Visual Studio .NET, the project by default contains a Views directory.

In an application, if a controller action returns a view, your application should:

- Have a folder for the controller, with the same name as the controller without the Controller suffix. For example, there must be a Home folder inside the Views folder, if an action method of the HomeController returns a view.
- Have a view file in the Home folder, with the same name as the action. For example, there must be an Index.cshtml view in the Home folder, if the Index action of the HomeController returns a view.

Code Snippet 1 shows an `Index` action that returns an `ActionResult` object through a call to the `View()` method of the Controller class.

Code Snippet 1:

```
public class HomeController : Controller {
    public ActionResult Index() {
        return View();
    }
}
```

Code Snippet 1 shows the `Index` action of the controller named, `HomeController` that returns the result of a call to the `View()` method. The result of the `View()` method is an `ActionResult` object that renders a view.

Visual Studio .NET simplifies the process of creating a view for a controller action. Instead of manually creating the folder structure and adding a view file, you can create the view file by performing the following steps:

1. Right-click inside the action method for which you need to create a view.
2. Select **Add View** from the context menu that appears. The **Add View** dialog box appears.

Figure 3.1 shows the **Add View** dialog box.

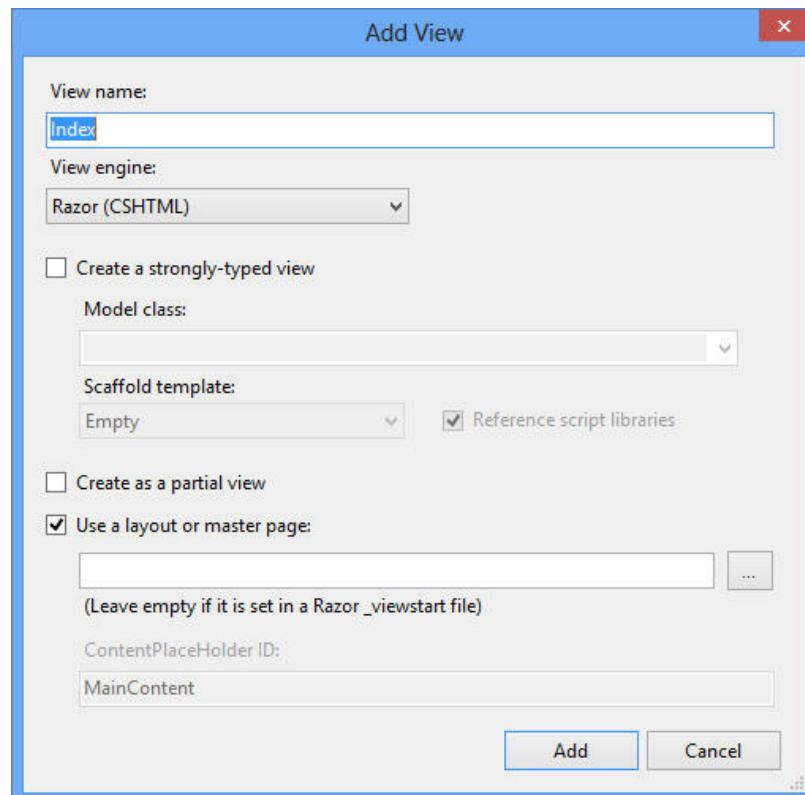


Figure 3.1: Add View Dialog Box

3. Click **Add**. Visual Studio .NET automatically creates the appropriate directory structure and adds the view file to it.

Figure 3.2 shows the view file that Visual Studio .NET creates for the `Index` action method of the `HomeController` class in the **Solution Explorer** window.

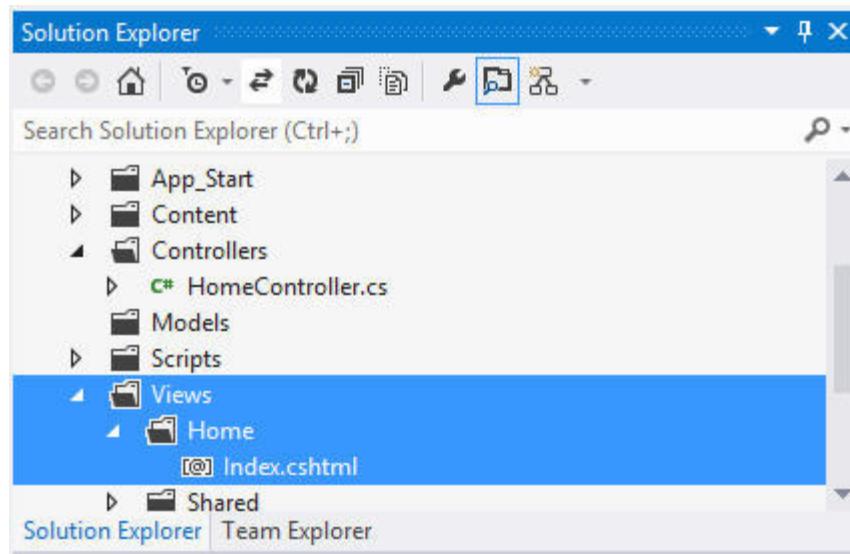


Figure 3.2: Solution Explorer Window

In the `Index.cshtml` file, you can add the content that the view should display.

Code Snippet 2 shows the content of the `Index.cshtml` view.

Code Snippet 2:

```
<!DOCTYPE html>
<html>
<head>
<title>Test View</title>
</head>
<body>
<h1>Welcome to the Website</h1>
</body>
</html>
```

The code creates a view with a title and a message as a heading.

When you access the `Index` action of the `HomeController` from a browser, the `Index.cshtml` view will be displayed. Following URL accesses the `Index` action of the `HomeController`.

`http://localhost:1267/Home/Index`

Note - The port number in the preceding URL might vary each time you run the application in Internet Information Services (IIS) Express embedded in Visual Studio .NET.

Figure 3.3 shows the `Index.cshtml` view rendered in the browser.

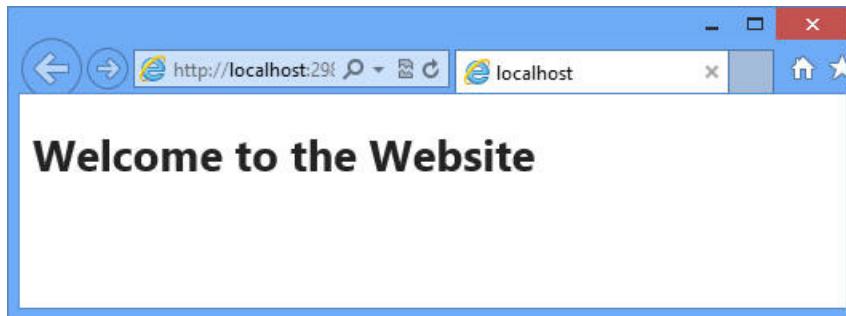


Figure 3.3: Index.cshtml View

You can also render a different view from an action method. To return a different view, you need to pass the name of the view as a parameter.

Code Snippet 3 shows how to return a different view.

Code Snippet 3:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View("TestIndex");
    }
}
```

The code will search for a view inside the `/Views/Home` folder, but render the `TestIndex` view instead of rendering the `Index` view.

While developing an ASP.NET MVC application, you might also need to render a view that is present in a different folder instead of the default folder. For example, the `Index` action of the `HomeController` class might need to display a view named `Welcome.cshtml` present in the `/Views/Demo` folder. To render such view, you need to specify the path to the view.

Code Snippet 4 shows displaying a view named `Index.cshtml` present in the `/Views/Demo` folder.

Code Snippet 4:

```
public class HomeController : Controller
{
}
```

```
public ActionResult Index()
{
    return View("~/Views/Demo/Welcome.cshtml");
}
```

The code displays the view, named `Welcome.cshtml` defined inside the **/Views/Demo** folder.

3.1.3 Passing Data from a Controller to a View

In an ASP.NET MVC application, a controller typically performs the business logic of the application and needs to return the result to the user through a view. So, a mechanism is required to pass the data from a controller to a view. You can use the following objects to pass data between controller and view:

- ➔ `ViewData`
- ➔ `ViewBag`
- ➔ `TempData`
- ➔ **`ViewData`**

You can use a `ViewData` object pass data from a controller to a view. `ViewData` is a dictionary of objects that is derived from the `ViewDataDictionary` class. The objects in `ViewData` are stored as key-value pairs that are accessible through the keys.

Some of the characteristics of `ViewData` are as follows:

- The life of a `ViewData` object exists only during the current request.
- The value of `ViewData` becomes null if the request is redirected.
- `ViewData` requires typecasting when you use complex data type to avoid error.

While using `ViewData`, you can specify values by using key-value pairs in the action method.

Syntax:

`ViewData[<key>] = <Value>;`

where,

Key: Is a String value to identify the object present in `ViewData`.

Value: Is the object present in `ViewData`. This object may be a String or a different type, such as `DateTime`.

Code Snippet 5 shows a ViewData with two key-value pairs in the Index action method of the HomeController class.

Code Snippet 5:

```
public class HomeController : Controller {
    public ActionResult Index() {
        ViewData["Message"] = "Message from ViewData";
        ViewData["CurrentTime"] = DateTime.Now;
        return View();
    }
}
```

In this code, a ViewData is created with two key-value pairs. The first key named, Message contains the String value, "Message from ViewData". The second key named, CurrentTime contains the value, DateTime.Now. Following is the syntax of ViewData:

Syntax:

@ViewData[<key>]

where,

Key: Is the value used to retrieve the corresponding value present in ViewData.

Code Snippet 6 shows retrieving the values present in ViewData.

Code Snippet 6:

```
<!DOCTYPE html>
<html>
<head>
<title>Index View</title>
</head>
<body>
<p>
    @ViewData["Message"]
</p>
<p>
    @ViewData["CurrentTime"]
</p>
```

```
</p>
</body>
</html>
```

In Code Snippet 6, `ViewData` is used to display the values of the `Message` and `CurrentTime` keys.

Figure 3.4 shows the output of the `ViewData`.

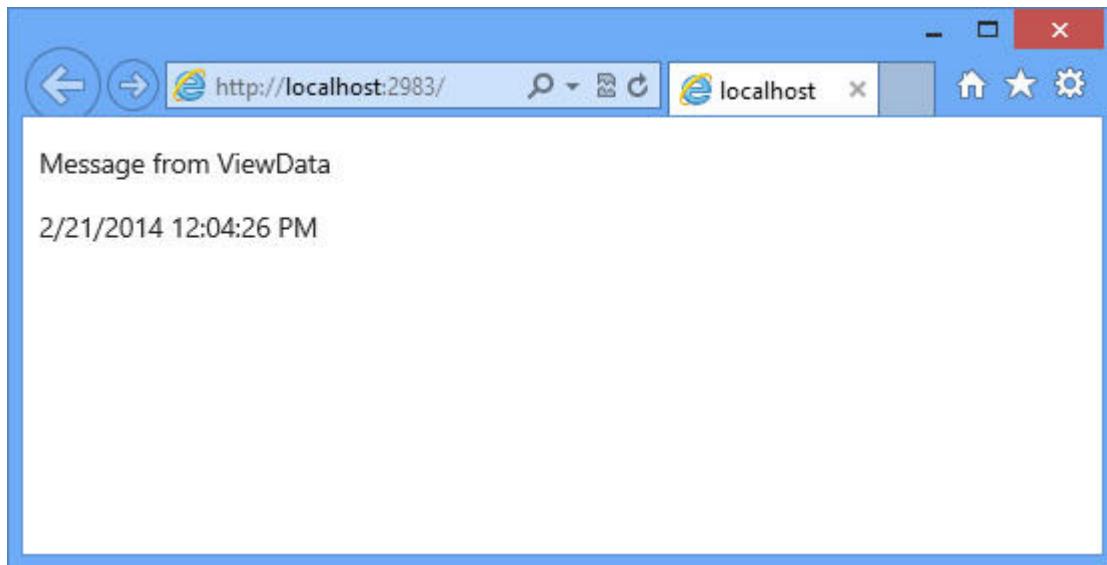


Figure 3.4: Output of `ViewData`

→ **ViewBag**

In the view, you can access and display the `ViewBag` data. `ViewBag` is a wrapper around `ViewData` and similar to `ViewData`, `ViewBag` exists only for the current request and becomes null if the request is redirected. However, unlike `ViewData`, which is a dictionary of key-value pairs, `ViewBag` is a dynamic property, based on the dynamic features introduced in C# 4.0. In addition, unlike `ViewData`, `ViewBag` does not require typecasting when you use complex data type.

Syntax:

```
ViewBag.<Property> = <Value>;
```

where,

Property: Is a String value that represents a property of `ViewBag`.

Value: Is the value of the property of `ViewBag`.

Code Snippet 7 shows a ViewBag with two properties in the Index action method of the HomeController class.

Code Snippet 7:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message = "Message from ViewBag";
        ViewBag.CurrentTime = DateTime.Now;
        return View();
    }
}
```

In this code, a ViewBag is created with two properties. The first property named, Message contains the String value, “Message from ViewBag”. The second property named, CurrentTime contains the value, DateTime.Now.

Code Snippet 8 shows retrieving the values present in ViewBag.

Code Snippet 8:

```
<!DOCTYPE html>
<html>
<head>
<title>Index View</title>
</head>
<body>
<p>
    @ViewBag.Message
</p>
<p>
    @ViewBag.CurrentTime
</p>
</body>
</html>
```

In this code, ViewBag is used to display the values of the Message and CurrentTime properties. Figure 3.5 shows the output of ViewBag.

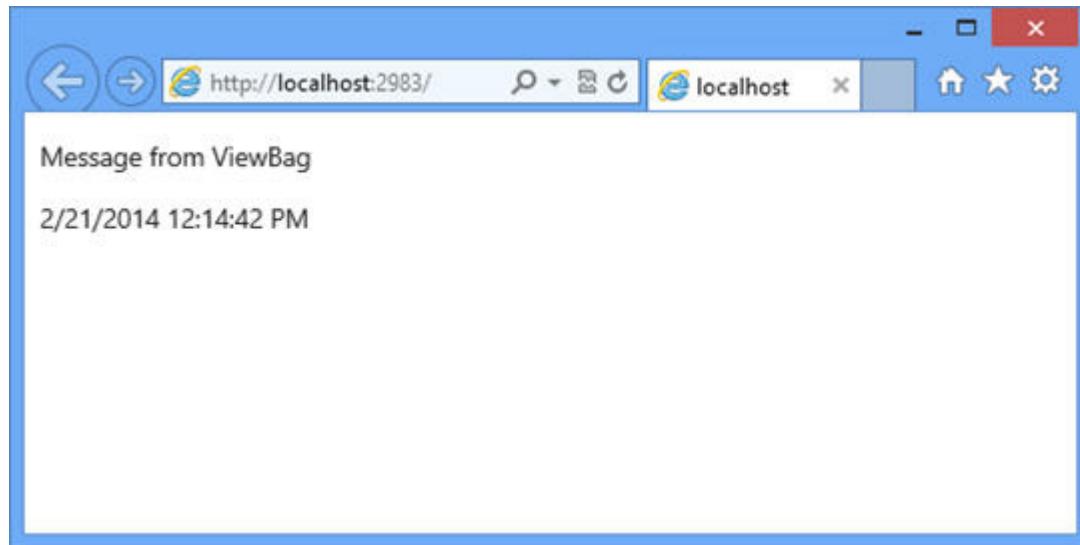


Figure 3.5: Output of ViewBag

When you use a ViewBag to store a property and its value in an action, that property can be accessed by both ViewBag and ViewData in a view.

Code Snippet 9 shows a controller action storing a ViewBag property.

Code Snippet 9:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.CommonMessage = "Common message accessible to both ViewBag and
ViewData";
        return View();
    }
}
```

In this code, a ViewBag is created with a property named, CommonMessage.

Code Snippet 10 shows a view that uses both ViewData and ViewBag to access the CommonMessage property stored in ViewBag.

Code Snippet 10:

```
<!DOCTYPE html>
<html>
<head>
<title>Index View</title>
</head>
<body>
<p>
<em>Accessed from ViewData:</em> @ViewData["CommonMessage"]
</p>
<p>
<em>Accessed from ViewBag:</em> @ViewBag.CommonMessage
</p>
</body>
</html>
```

The code uses both ViewData and ViewBag to display the value of the CommonMessage property stored in ViewBag.

Figure 3.6 shows the output of ViewData and ViewBag.

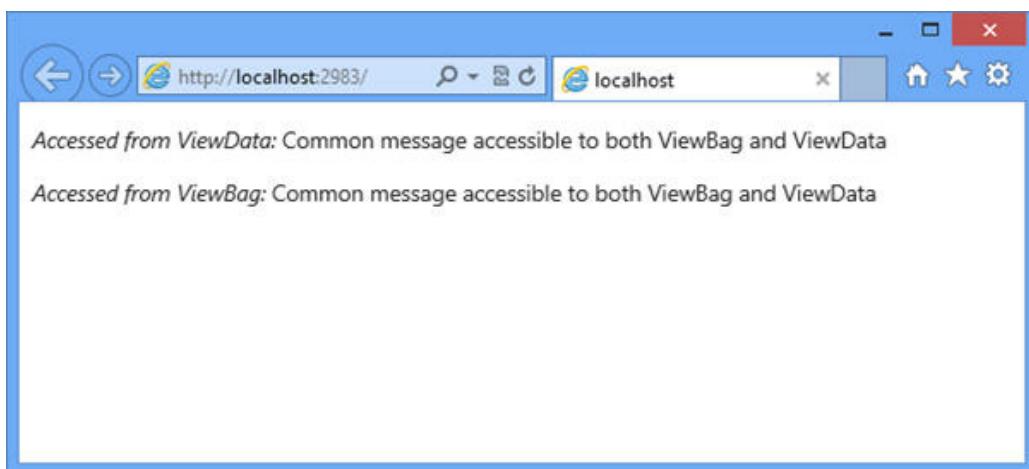


Figure 3.6: Output of ViewData and ViewBag

→ **TempData**

Another object that you can use to pass data from a controller to a view is TempData. TempData is a Dictionary object derived from the TempDataDictionary class. Similar to ViewData, TempData stores data as key-value pairs. However, unlike ViewData and ViewBag that does not preserve values during request redirections, TempData allows passing data from the current request to the subsequent request during request redirection.

Syntax:

TempData[<Key>] = <Value>;

where,

Key: Is a String value to identify the object present in TempData.

Value: Is the object present in TempData.

Code Snippet 11 shows how to use TempData to pass values from one view to another view through request redirection.

Code Snippet 11:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewData["Message"] = "ViewData Message";
        ViewBag.Message = "ViewBag Message";
        TempData["Message"] = "TempData Message";
        return RedirectToAction("Home/About");
    }

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

The code creates two actions, Index and About in the HomeController class. The Index action stores value to ViewData, ViewBag, and TempData objects. The Index action then redirects the request to the About action by calling the RedirectToAction() method. The About action returns the corresponding view, which is the About.cshtml view.

Code Snippet 12 shows the About.cshtml view.

Code Snippet 12:

```
<!DOCTYPE html>
<html>
<head>
<title>Index View</title>
</head>
<body>
<p>
<em>Accessed from ViewData:</em> @ViewData["Message"]
</p>
<p>
<em>Accessed from ViewBag:</em> @ViewBag.Message
</p>
<p>
<em>Accessed from TempData:</em> @TempData["Message"]
</p>
</body>
</html>
```

The code shows the About.cshtml view that the About action returns. This view attempts to access the values stored in the ViewData, ViewBag, and TempData objects.

Figure 3.7 shows the output of About.cshtml view.

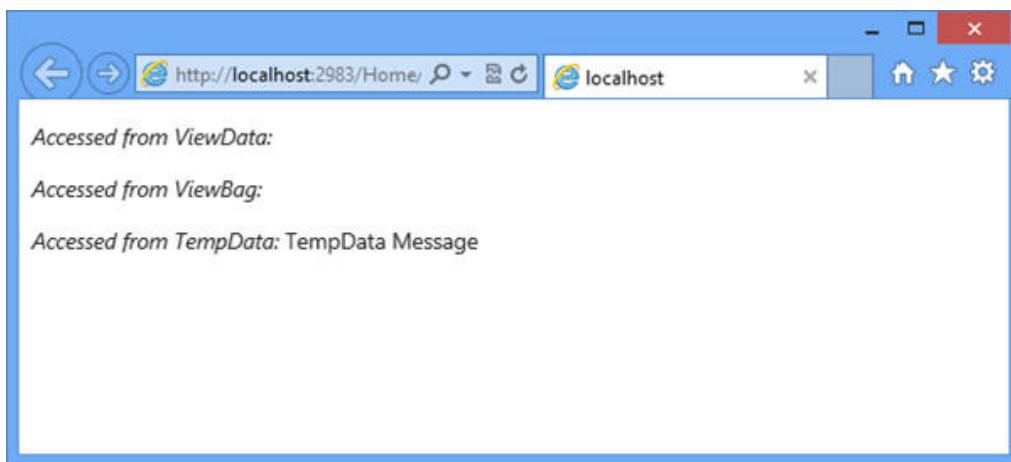


Figure 3.7: Output of About.cshtml View

In figure 3.7, note that the values stored in the `ViewData` and `ViewBag` objects are not displayed. This is because both the objects do not preserve values during the request redirection process. However, as `TempData` preserves values during request redirections, the value stored in `TempData` is displayed.

3.1.4 Using Partial Views

In an ASP.NET MVC application, a partial view represents a sub-view of a main view. Partial views allow you to reuse common markups across the different views of the application. For example, in an online shopping cart application, you can create a partial view that displays the different available categories of products. You can then use the partial view in all the views of the application that needs to display the categories.

Note - By convention, the name of a partial view is prefixed with an underscore (_) character and stored in the **Views/Shared** folder in the application directory structure.

To create a partial view in Visual Studio .NET, you need to perform the following steps:

1. Right-click the **Views/Shared** folder in the **Solution Explorer** window and select **Add→View**. The **Add View** dialog box appears.
2. In the **Add View** dialog box, specify a name for the partial view in the **View Name** text field.
3. Select the **Create as a partial view** check box.

Figure 3.8 shows how to create a partial view.

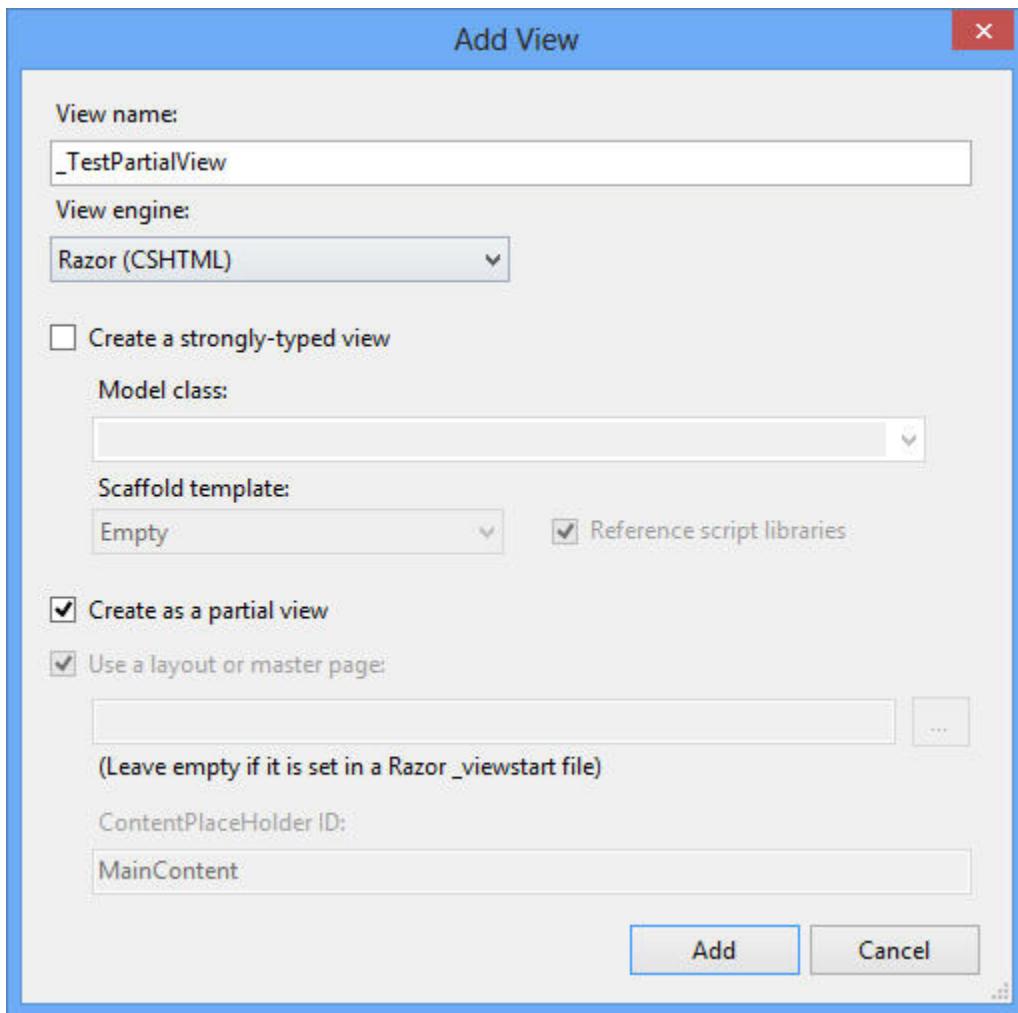


Figure 3.8: Creating a Partial View

- Click **Add** to create the partial view.

In the partial view, add the markup that you need to display in the main view, as shown in the following code:

```
<h3> Content of partial view. </h3>
```

Syntax:

```
@Html.Partial(<partial_view_name>)
```

where,

`partial_view_name`: Is the name of the partial view without the `.cshtml` file extension.

Code Snippet 13 shows a main view, named `Index.cshtml` that accesses the partial view, named, `_TestPartialView.cshtml`.

Code Snippet 13:

```
<!DOCTYPE html>
<html>
<head>
<title>Index View</title>
</head>
<body>
<h1>
    Welcome to the Website
</h1>
<div>@Html.Partial("_TestPartialView")</div>
</body>
</html>
```

The code shows the markup of the main `Index.cshtml` view that displays a welcome message and renders the partial view named, `_TestPartialView.cshtml`.

Figure 3.9 shows the output of the main view named, `Index.cshtml`.

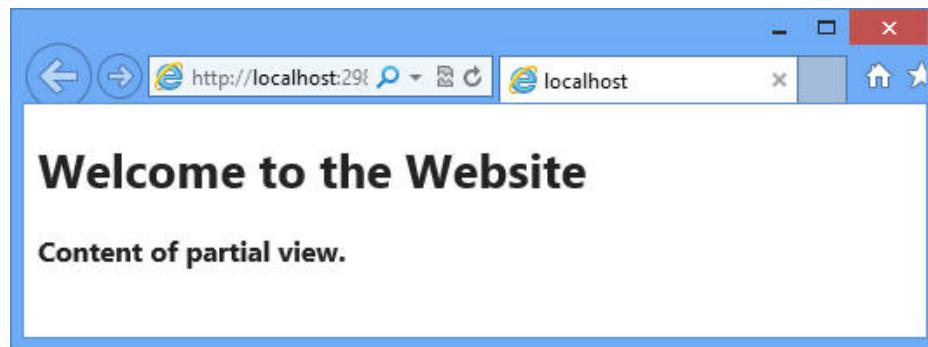


Figure 3.9: Output of `Index.cshtml`

3.2 Razor

Razor is the syntax, based on the ASP.NET Framework that allows you to create views. Razor combines HTML markups for presenting a view and programming code, such as Visual C#.NET or Visual Basic (VB).NET to implement the logic of the view. Razor is simple and easy to understand and with its minimum number of constructs does not introduce a learning curve for users who are familiar with the C#.NET or VB.NET programming languages.

Code Snippet 14 shows a simple Razor view that contains both HTML markups for presentation and C# code to implement the view logic.

Code Snippet 14:

```
@{  
  
var products = new string[] {"Laptop", "Desktop", "Printer"};  
}  
  
<html>  
<head><title>Test View</title></head>  
<body>  
<ul>  
@foreach (var product in products)  
{  
<li>The product is @product.</li>  
}  
</ul>  
</body>  
</html>
```

In this code, a `string[]` is declared and initialized using Razor syntax. Then, Razor syntax is used to iterate through the elements of the array and display each element. The remaining code in the view is HTML code that displays the page title, body, and the array elements in a bullet list.

Figure 3.10 shows the output of simple Razor view.

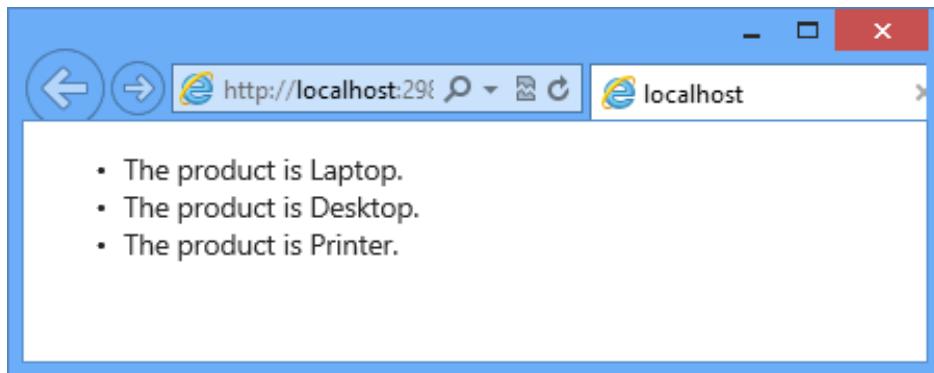


Figure 3.10: Output of Simple Razor View

3.2.1 Razor Engine

The MVC Framework uses a view engine to convert the code of a view into HTML markup that a browser can understand. The MVC Framework uses Razor as the default view engine. The Razor view engine compiles a view of your application when the view is requested for the first time. The Razor view engine then delivers the compiled view for subsequent requests until you make changes to the view.

The Razor view engine does not introduce a new set of programming language, but provides template markup syntax to segregate HTML markup and programming code in a view. As a result, developers familiar with the C# and VB programming languages can easily create views that target the Razor engine.

In addition, the Razor engine supports Test Driven Development (TDD) which allows you to independently test the views of an application.

3.2.2 Razor Syntax Rules

To interpret the server-side code embedded inside a view file, Razor first requires identifying the server-side code from the markup code. Razor uses the @ symbol to separate the server-side code from the markup code.

While creating a Razor view, you should consider the following rules:

- Enclose code blocks between @{} and {}
- Start inline expressions with @
- Variables are declared with the var keyword
- Enclose strings in quotation marks
- End a Razor code statement with semicolon (;)
- Use the .cshtml extension to store a Razor view file that uses C# as the programming language
- Use the .vbhtml extension to store a Razor view file that uses VB as the programming language
- **Including Code Blocks**

Razor supports code blocks within a view. A code block is a part of a view that only contains code written in C# or VB.

Syntax:

```
@{  
    <code>  
}
```

where,

`code`: Is the C# or VB code that will execute on the server.

Code Snippet 15 shows two single-statement code blocks in a Razor view.

Code Snippet 15:

```
@{ var myMessage = "HelloWorld"; }
{@{ var num = 10; }}
```

The code shows two single-statement code blocks that declares the variables, `myMessage` and `num`. The `@{` characters mark the beginning of each code block and the `}` character marks the end of the code blocks.

Razor also supports multi-statement code blocks where each code block can have multiple statements. Multi-statement code block allows you to ignore the use of the `@` symbol in every line of code.

Code Snippet 16 shows a multi-statement code block in a Razor view.

Code Snippet 16:

```
@{
    var myMessage = "HelloWorld";
    var num = 10;
}
```

This code shows a multi-statement code block that declares the variables, `myMessage` and `num`.

Note - The code that you use in a code block is regular C# and VB code that must be following the programming rules of the language. For example, each C# statement in a code block is case sensitive and must end with a semicolon.

→ Including Inline Expressions

Similar to code block, Razor uses the `@` character for an inline expression. Code Snippet 17 shows using inline expressions.

Code Snippet 17:

```
@{
    var myMessage = "HelloWorld";
    var num = 10;
}
@myMessage is numbered @num.
```

The code uses two inline expressions that evaluates the `myMessage` and `num` variables and outputs the result of the expression. When the Razor engine encounters the `@` character, it interpreted the immediately following variable name as server-side code and ignores the following text. Therefore, you are not requiring specifying the end of the Razor code for inline expressions.

Figure 3.11 shows the output of Code Snippet 17.

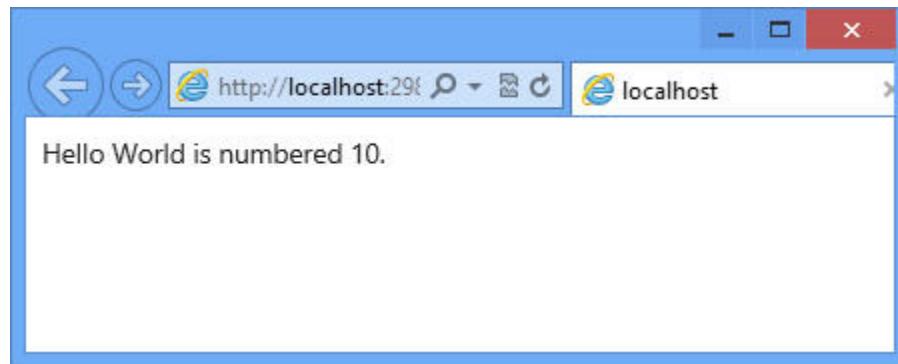


Figure 3.11: Output of Inline Expressions

At times, you may also require overriding the logic that Razor uses to identify the server-side code. For example, whenever you requires displaying the `@` symbol within an e-mail address, you can use `@@`.

Code Snippet 18 shows including the `@`symbol in an e-mail address.

Code Snippet 18:

```
<h3>The email ID is: john@@mvceexample.com </h3>
```

The code Razor interprets the first `@` symbol as an escape sequence character. To identify the part of a server side code, Razor uses an implicit code expression. Sometimes, Razor may also interpret an expression as markup instead of running as server-side code.

Figure 3.12 shows the output of including `@` symbol.

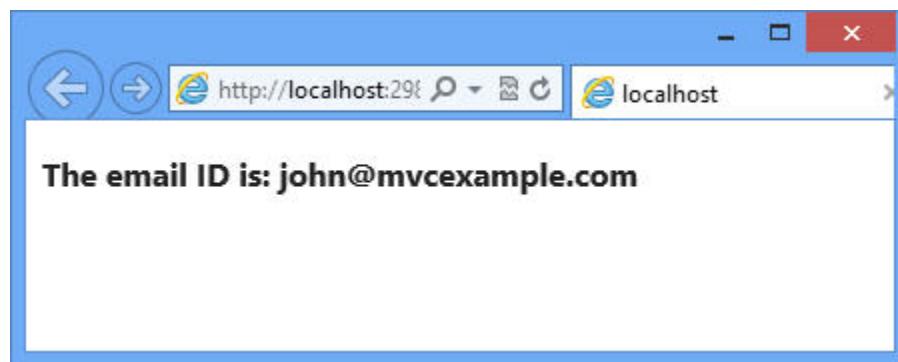


Figure 3.12: Output of Including @ Symbol

→ Including Comments

You can declare a Razor comment within the @* and *@ delimiters.

Syntax:

```
@* This is a comment in a view. *@
```

3.2.3 Variables

Variables are used to store data. In Razor, you declare and use a variable in the same way as you do in a C# program.

Code Snippet 19 shows declaration of variables using Razor.

Code Snippet 19:

```
<!DOCTYPE html>
<html>
<body>
@{
    var heading = "Using variables";
    string greeting = "Welcome ASP.NET MVC Application";
    int num = 103;
    DateTime today = DateTime.Today;
}
<h3>@heading</h3>
<p>@greeting</p>
<p>@num</p>
<p>@today</p>
</body>
</html>
```

The code shows declaring four variables, such as heading, greeting, num, and today using Razor.

Figure 3.13 shows the output of declaring variables using Razor.

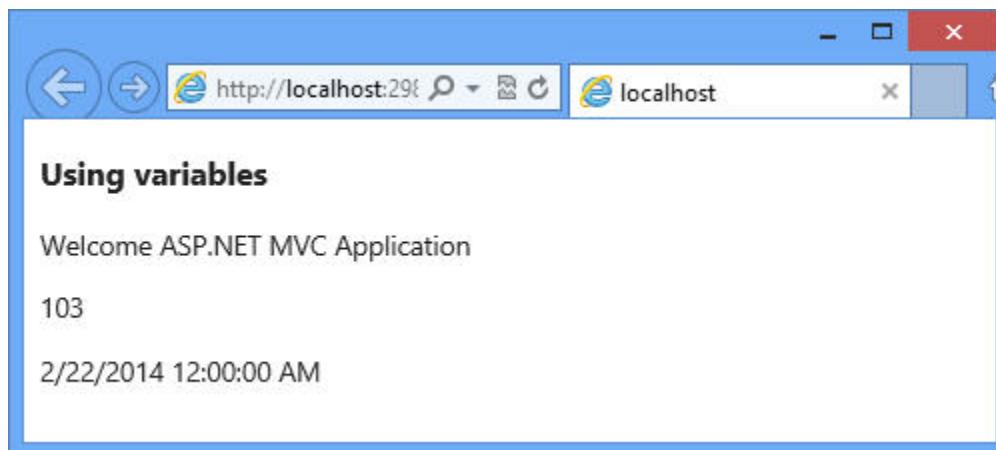


Figure 3.13: Output of Declaring Variables Using Razor

3.2.4 Loops

While developing an ASP.NET MVC Web application, you might require executing same statement continually. In such scenario, you can use loops. C# supports four types of loop constructs. These are as follows:

- The `while` loop
- The `for` loop
- The `do...while` loop
- The `foreach` loop

The `while` loop is used to execute a block of code repetitively as long as the condition of the loop remains true. To use `while` loop, you need to use the `while` keyword at the beginning followed by parentheses. Inside the parentheses, you can specify the number of time the loop continues, then a block to repeat.

Code Snippet 20 shows the Razor code that uses a `while` loop to print the even numbers from 1 to 10.

Code Snippet 20:

```
<!DOCTYPE html>
<html>
<body>
@{
    var b = 0;
    while (b < 7)
```

```
{
    b += 1;
    <p>Text @b</p>
}
}
</body>
</html>
```

The code shows a `while` loop to print the even numbers from 1 to 10 using Razor.

Figure 3.14 shows the output of the razor code using `while` loop.

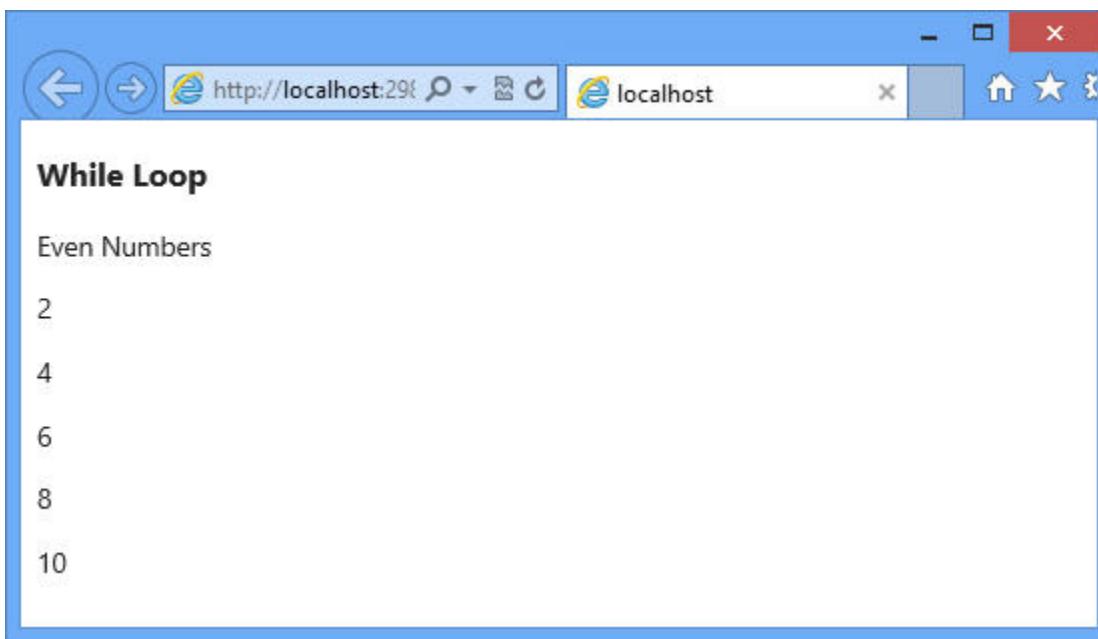


Figure 3.14: Razor Using `while` Loop

When you know the number of time that a statement should execute, you can use a `for` loop. Similar to the `while` loop, you can use the `for` loop to iterate through elements.

Code Snippet 21 shows the Razor code that uses a `for` loop to print the even numbers from 1 to 10.

Code Snippet 21:

```
<!DOCTYPE html>
<html>
<body>
<h1>Even Numbers</h1>
```

```
@{
    var num=1;

    for (num=1; num<=11; num++)
    {
        if ((num % 2) == 0)
        {
            <p> @num </p>
        }
    }
}

</body>
</html>
```

The code shows the `for` loop to print the even numbers from 1 to 10 using Razor.

Figure 3.15 shows the output of the Razor using `for` loop.

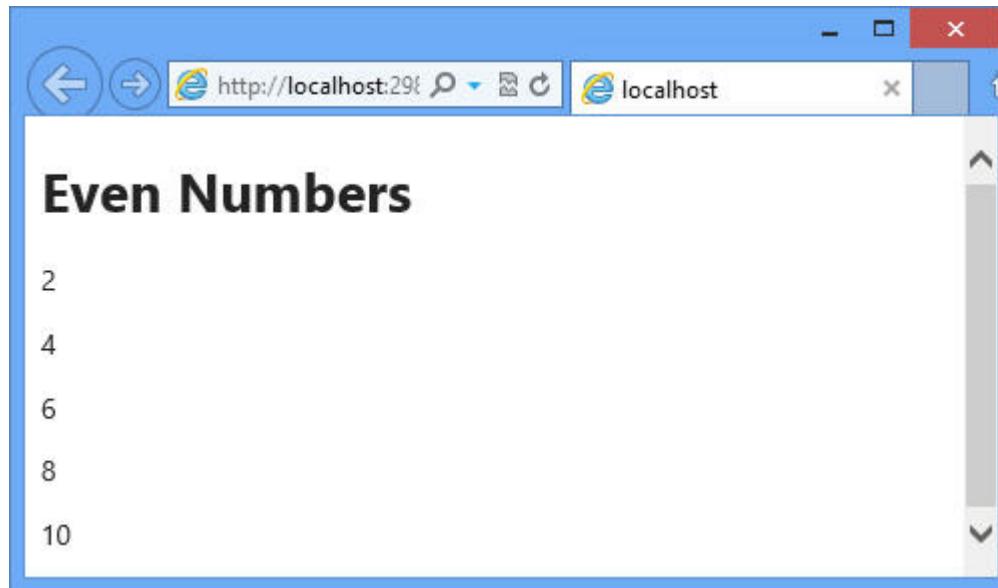


Figure 3.15: Razor Using for Loop

3.2.5 Conditional Statements

While developing an ASP.NET MVC application, sometime you may need to display dynamic content based on some specific conditions. In such cases, you can use conditional statements, such as the `if` statement. The `if` statement returns true or false, based on the specified condition. You can also use the `if` and `else` statements to generate dynamic content for your views.

Code Snippet 22 shows using the if ... else statements using Razor.

Code Snippet 22:

```
<!DOCTYPE html>
@{var mark=60; }

<html>
<body>
@if (mark>80)
{
<p>You have failed in the exam.</p>
}
else
{
<p>You have passed the exam.</p>
}
</body>
</html>
```

The code uses the if ... else statements using Razor.

Figure 3.16 shows the output of the Razor using if...else.

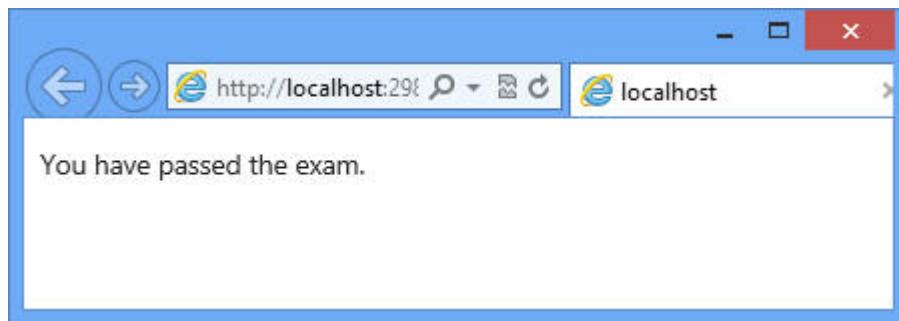


Figure 3.16: Razor Using if...else

You can also test a number of individual conditions in your code. To do this, you can use a switch statement in the view.

Code Snippet 23 shows using the `switch` statement using Razor.

Code Snippet 23:

```
<!DOCTYPE html>
@{
    var day=DateTime.Now.DayOfWeek.ToString();
    var msg="";
}
<html>
<body>
@switch(day)
{
    case "Monday":
        msg="Today is Monday, the first working day.";
        break;
    case "Friday":
        msg="Today is Friday, the last working day.";
        break;
    default:
        msg="Today is " + day;
        break;
}
<p>@msg</p>
</body>
</html>
```

The code shows the `switch` statement using Razor.

Figure 3.17 shows the output of the Razor using switch statement.

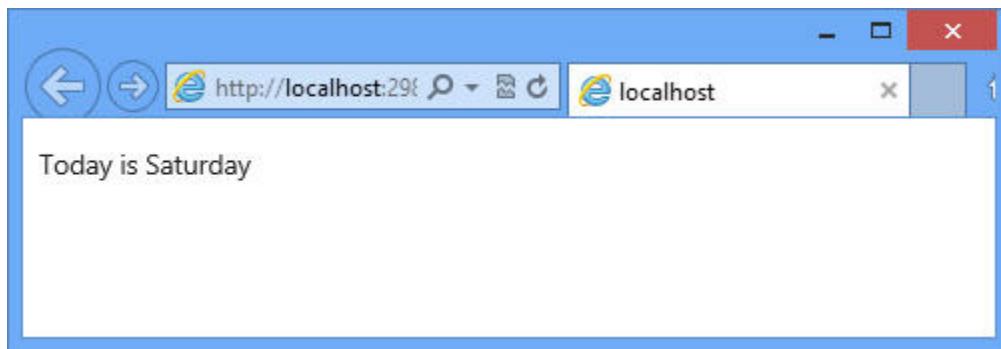


Figure 3.17: Razor Using switch Statement

3.2.6 Arrays

You can make use of arrays while writing your code. Using array is useful when you need to store similar variables. Thus, you can avoid creating a separate variable for each item.

Code Snippet 24 shows the arrays to store similar variables using Razor.

Code Snippet 24:

```
<!DOCTYPE html>
@{
    string[] members = {"Joe", "Mark", "Stella"};
    int i = Array.IndexOf(members, "Stella") + 1;
    int len = members.Length;
    string x = members[2 - 1];
}
<html>
<body>
<h3>Student Details</h3>
@foreach (var person in members)
{
    <p>@person</p>
}
<p>The number of students in class are @len</p>
<p>The student at position 2 is @x</p>
<p>Stella is now in position @i</p>
```

```
</body>  
</html>
```

The code shows the arrays that stores similar variables using Razor.

Figure 3.18 shows the output of arrays that stores similar variables.

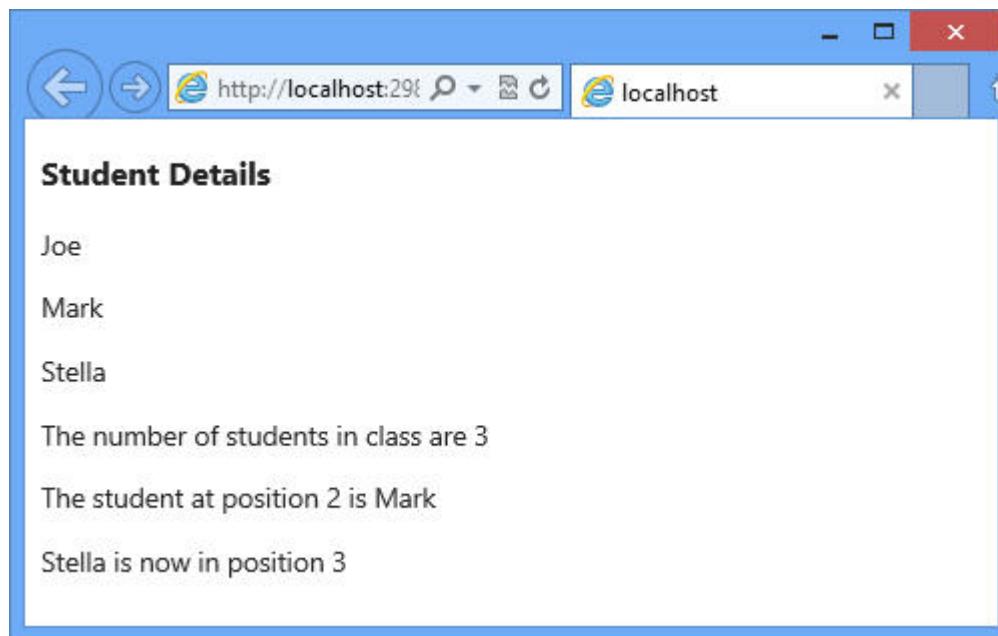


Figure 3.18: Using Arrays

3.3 HTML Helper Methods

The MVC Framework provides built-in HTML helper methods that simplify creating a view. You can use these methods to generate HTML markup that you can reuse across the Web application. These HTML helper methods, which are extension methods to the `HtmlHelper` class, can be called only from views.

Some of the commonly used helper methods while developing an MVC application are as follows:

- ➔ `Html.ActionLink()`
- ➔ `Html.BeginForm()` and `Html.EndForm()`
- ➔ `Html.Label()`
- ➔ `Html.TextBox()`
- ➔ `Html.TextArea()`
- ➔ `Html.Password()`

- `Html.CheckBox()`
- `Html.DropDownList()`
- `Html.RadioButton()`

3.3.1 Using `Html.ActionLink()`

The `Html.ActionLink()` helper method allows you to generates a hyperlink that points to an action method of a controller class.

Syntax:

```
@Html.ActionLink(<link_text>, <action_method>, <optional_controller>)
```

where,

`link_text`: Is the text to be displayed as a hyperlink.

`action_method`: is the name of the action method that acts as the target for the hyperlink.

`optional-controller`: Is the name of the controller that contains the action method that will get called by the hyperlink. This parameter can be omitted, if the action method getting called is in the same controller as the action method whose view renders the hyperlink.

Code Snippet 25 shows using an `Html.ActionLink()` helper method.

Code Snippet 25:

```
<!DOCTYPE html>
<html>
<body>
    @Html.ActionLink("Click to Browse", "Browse", "Home")
</body>
</html>
```

In the code, the `Click to Browse` is the text that is to be displayed as a hyperlink. The `Browse` action method of the `Home` controller acts as the target of the hyperlink.

Figure 3.19 shows the output of an `Html.ActionLink()` helper method.

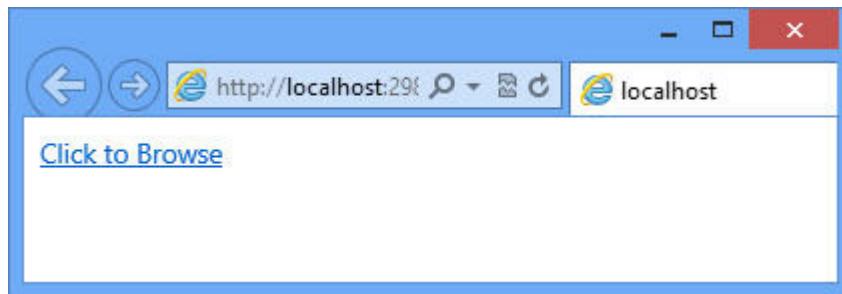


Figure 3.19: Using a `Html.ActionLink()` Helper Method

3.3.2 Using `Html.BeginForm()` and `Html.EndForm()`

The `Html.BeginForm()` helper method allows you to mark the start of a form. In order to generate a URL, this helper method coordinates with the routing engine. This helper method is responsible for producing the opening `<form>` tag.

Syntax:

```
@{Html.BeginForm(<action_method>,<controller_name>);}
```

where,

`action_method`: is the name of the action method.

`controller_name`: is the name of the controller class.

Once you used the `Html.BeginForm()` helper method to start a form, you need to end a form. You can use the `Html.EndForm()` helper method to end a form.

Code Snippet 26 shows using the `Html.BeginForm()` and `Html.EndForm()` helper methods.

Code Snippet 26:

```
<!DOCTYPE html>
<html>
<body>
@{Html.BeginForm("Browse","Home");}
<p>Inside Form</p>
@{Html.EndForm();}
</body>
</html>
```

In this code, the `Html.BeginForm()` method specifies the `Browse` action of the `Home` controller as the target action method to which the form data will be submitted.

Figure 3.20 shows the output of the `Html.BeginForm()` and `Html.EndForm()` helper methods.

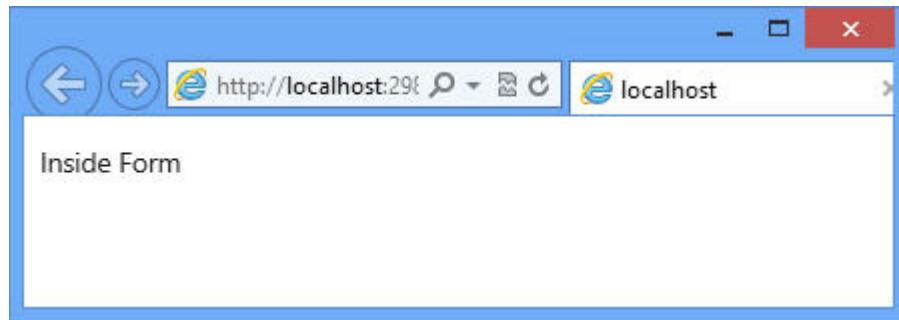


Figure 3.20: Using the `Html.BeginForm()` and `Html.EndForm()` Methods

You can avoid using the `Html.EndForm()` helper method to explicitly close the form by using the `@using` statement before the `Html.BeginForm()` method, as shown in the following Code Snippet 27.

Code Snippet 27:

```
@using (Html.BeginForm("NewAction", "NewController"))
{
    //Use markup to generate form elements
}
```

3.3.3 Using `Html.Label()`

You can use the `Html.Label()` helper method to display a label in a form. The purpose of a label is to attach information to other input elements, such as text inputs and increase the accessibility of your application.

Syntax:

`@Html.Label(<label_text>)`

where,

`label_text`: Is the name of the label.

Code Snippet 28 shows using the `Html.Label()` helper method.

Code Snippet 28:

```
@Html.Label("name")
<!DOCTYPE html>
```

```
<html>
<body>
@{Html.BeginForm("Browse", "Home");}
@Html.Label("User Name:")
@{Html.EndForm();}
</body>
</html>
```

In this code, the `Html.Label()` method creates a label with User Name as its name.

Figure 3.21 shows the output of the `Html.Label()` helper method.

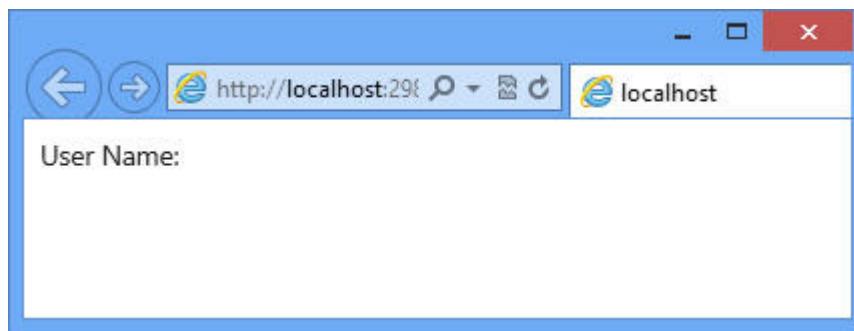


Figure 3.21: Using the `Html.Label()` Helper Method

3.3.4 Using `Html.TextBox()`

You can use the `Html.TextBox()` helper method to display an input tag. While using this method, you should specify the type attribute as text. This helper method is used to accept input from a user.

Syntax:

```
@Html.TextBox("textbox_text")
```

where,

`textbox_text`: Is the name of the text box.

Code Snippet 29 shows using the `Html.TextBox()` helper method.

Code Snippet 29:

```
<!DOCTYPE html>
<html>
<body>
@{Html.BeginForm("Browse", "Home");}

```

```
@Html.Label("User Name :")</br>
@Html.TextBox("textBox1")</br></br>
<input type="submit" value="Submit">
@{Html.EndForm(); }
</body>
</html>
```

In this code, the `Html.TextBox()` method creates a text box with `textBox1` as its name.

Figure 3.22 shows the output of the `Html.TextBox()` helper method.

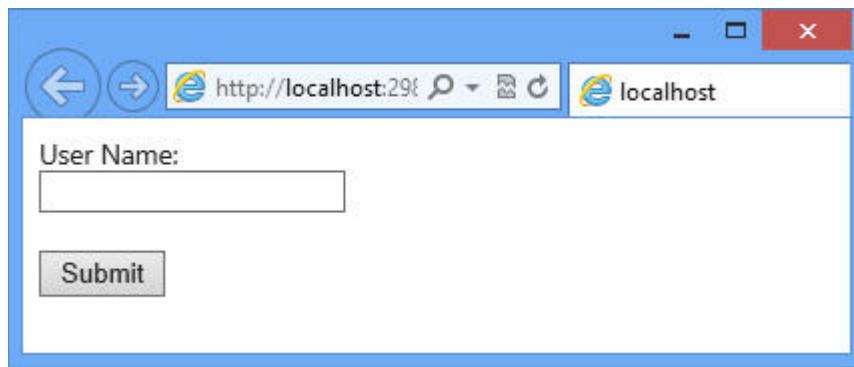


Figure 3.22: Using the `Html.TextBox()` Helper Method

3.3.5 Using `Html.TextArea()`

You can use the `Html.TextArea()` helper method to display a `<textarea>` element for multi-line text entry. This method enables you to specify the number of columns and rows to be displayed in order to control the size of the text area.

Syntax:

```
@Html.TextArea(<textarea_name>)
```

where,

`textarea_name`: Specifies the name of the text area.

Code Snippet 30 shows use of the `Html.TextArea()` helper method.

Code Snippet 30:

```
<!DOCTYPE html>
<html>
<body>
@{Html.BeginForm("Browse", "Home");}

```

```

@Html.Label("User Name:")</br>
@Html.TextBox("textBox1")</br></br>
@Html.Label("Address:")</br>
@Html.TextArea("textArea1")</br></br>
<input type="submit" value="Submit">
@{ Html.EndForm(); }
</body>
</html>

```

In Code Snippet 30, the `Html.TextArea()` method creates a text area with `textArea1` as its name.

Figure 3.23 shows the output of the `Html.TextArea()` helper method.

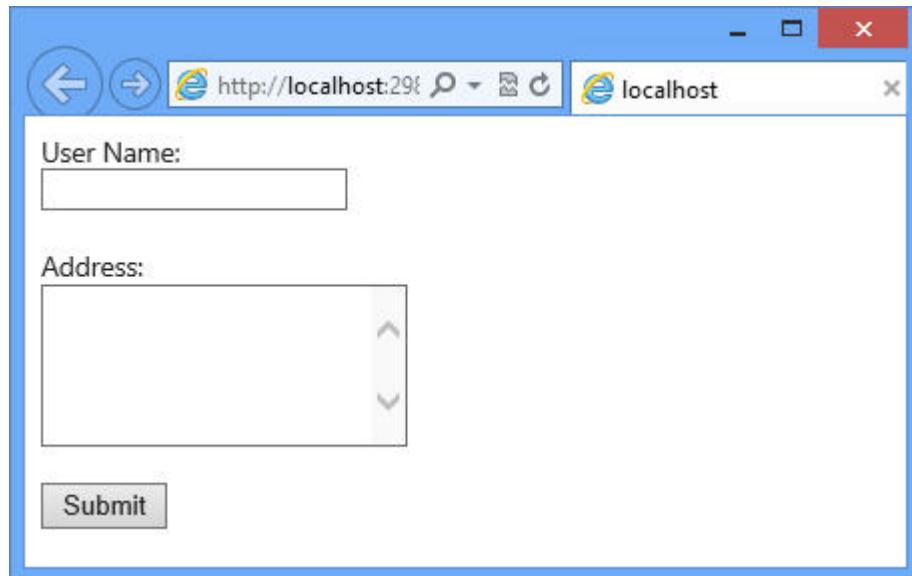


Figure 3.23: Using the `Html.TextArea()` Helper Method

3.3.6 Using `Html.Password()`

You can use the `Html.Password()` helper method to display a password field.

Syntax:

```
@Html.Password(<property_name>)
```

where,

`property_name`: Specifies the name of the password field.

Code Snippet 31 shows use of the `Html.Password()` helper method.

Code Snippet 31:

```
<!DOCTYPE html>
<html>
<body>
@{ Html.BeginForm("Browse", "Home"); }
@Html.Label("User Name:")<br>
@Html.TextBox("textBox1")<br><br>
@Html.Label("Address:")<br> @Html.TextArea("textareal")<br><br>
@Html.Label("Password:")<br>@Html.Password("password")<br><br>
<input type="submit" value="Submit">
@{ Html.EndForm(); }
</body>
</html>
```

In this code, the `Html.Password()` method creates a password field in the form with `password` as its name.

Figure 3.24 shows the output of the `Html.Password()` helper method.

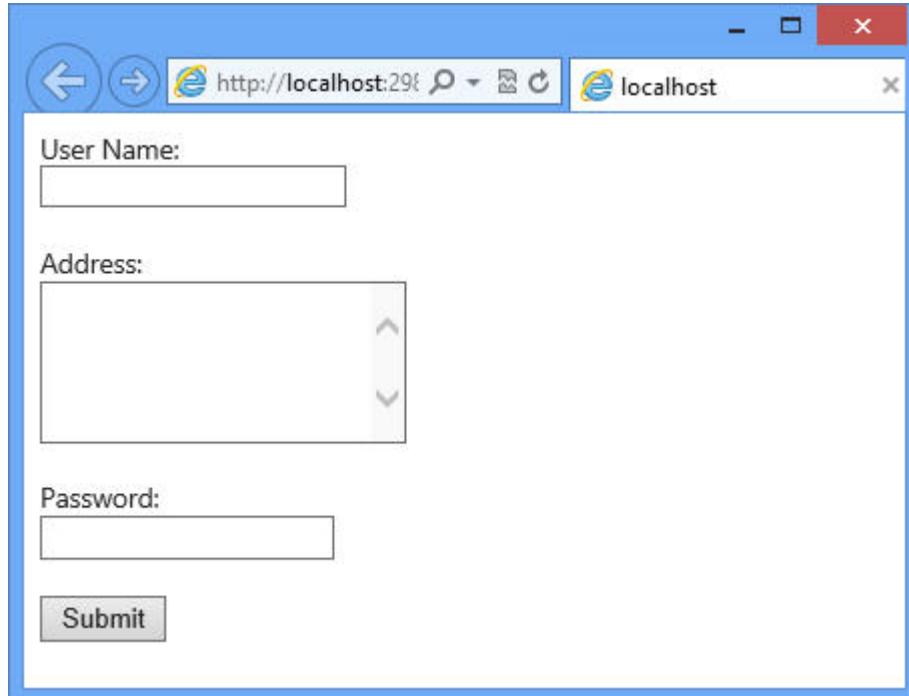


Figure 3.24: Using the `Html.Password()` Helper Method

3.3.7 Using `Html.CheckBox()`

You can use the `Html.CheckBox()` helper method to display a check box. The check box input element enables the user to select a true or false condition.

Syntax:

```
@Html.CheckBox("property_name")
```

where,

`property_name`: Specifies the name of the check box.

Code Snippet 32 shows use of the `Html.CheckBox()` helper method.

Code Snippet 32:

```
<!DOCTYPE html>
<html>
<body>
@{Html.BeginForm("Browse", "Home");}
@Html.Label("User Name:")
@Html.TextBox("textBox1")
@Html.Label("Address:")
@Html.TextArea("textArea1")
@Html.Label("Password:")
@Html.Password("password")
@Html.Label("I need updates on my mail:")
@Html.CheckBox("checkbox1")
<input type="submit" value="Submit">
@{Html.EndForm();}
</body>
</html>
```

In this code, the `Html.CheckBox()` helper method renders a hidden input in addition to the check box input. When a check box is selected by users, the browser submits a value for the check box. The hidden input ensures that a value will be submitted, even if the user does not select the check box.

Figure 3.25 shows the output of the `Html.CheckBox()` helper method.

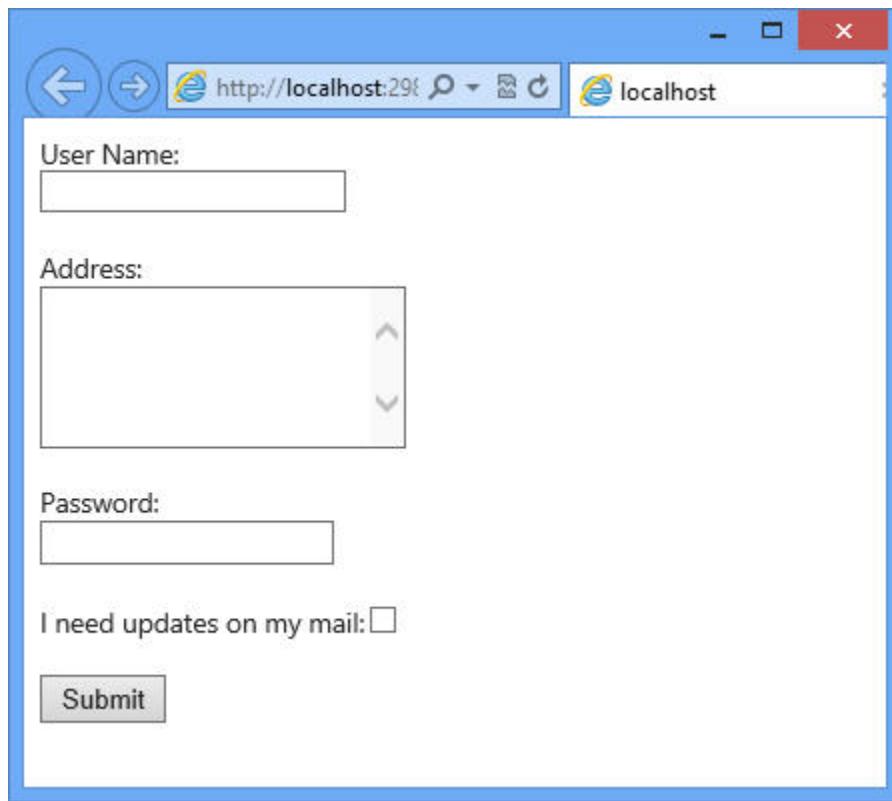


Figure 3.25: Using the `Html.CheckBox()` Helper Method

3.3.8 Using `Html.DropDownList()`

You can use the `Html.DropDownList()` helper method to return a `<select>` element. The `Html.DropDownList()` helper method allows selection of a single item. The `<select>` element shows a list of possible options and also the current value for a field.

Syntax:

```
@Html.DropDownList("myList", new SelectList(new [] {<value1>, <value2>, <value3>}), "Choose")
```

where

`value1`, `value2`, and `value3` are the options available in the drop-down list.

`Choose`: Is the value at the top of the list.

Code Snippet 33 shows use of the `Html.DropDownList()` helper method.

Code Snippet 33:

```
<!DOCTYPE html>
<html>
```

```

<body>
@{Html.BeginForm("Browse", "Home");}
@Html.Label("User Name:")
@Html.TextBox("textBox1")
@Html.Label("Address:")
@Html.TextArea("textArea1")
@Html.Label("Password:")
@Html.Password("password")
@Html.Label("I need updates on my mail:")
@Html.CheckBox("checkbox1")
@Html.Label("Select your city:")
@Html.DropDownList("myList", new SelectList(new [] {"New York", "Philadelphia", "California"}), "Choose")
<input type="submit" value="Submit">
@{Html.EndForm();}
</body>
</html>

```

In this code, the `Html.DropDownList()` method creates a drop-down list in a form with `myList` as its name and contains three values that the user can select from the drop-down list.

Figure 3.26 shows the output of the `Html.DropDownList()` helper method.

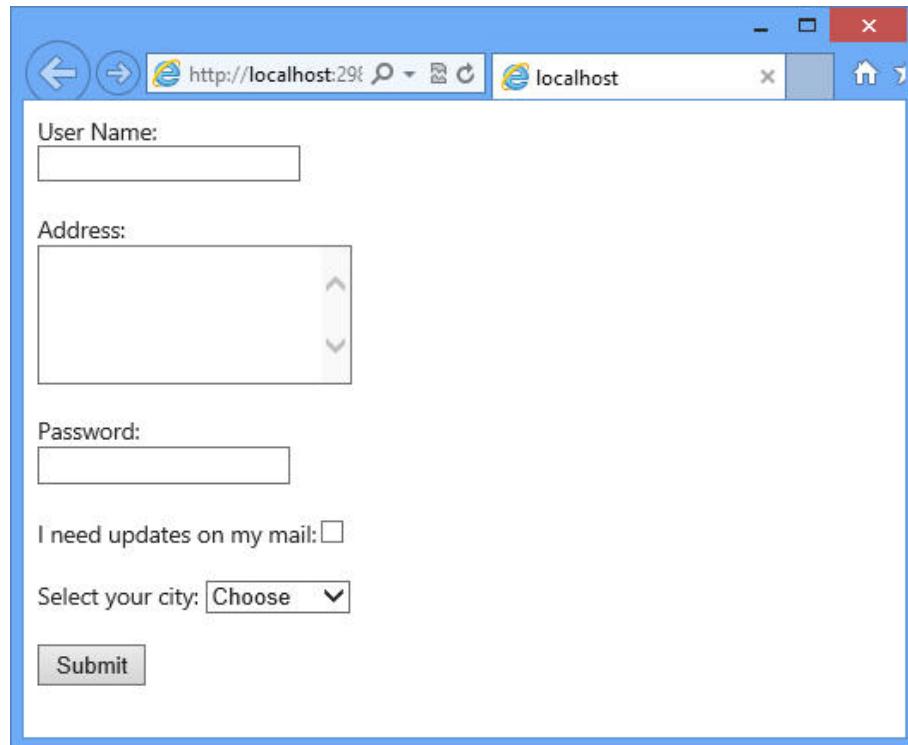


Figure 3.26: Using the `Html.DropDownList()` Helper Method

3.3.9 Using `Html.RadioButton()`

You can use the `Html.RadioButton()` helper method to provide a range of possible options for a single value. For example, if you want the user to select the gender, you can use two radio buttons to present the choices.

Syntax:

```
@Html.RadioButton("name", "value", isChecked)
```

where,

`name`: Is the name of the radio button input element.

`value`: Is the value associated with a particular radio button option.

`isChecked`: Is a Boolean value that indicates whether the radio button option is selected or not.

Code Snippet 34 shows use of the `Html.RadioButton()` helper method.

Code Snippet 34:

```
<!DOCTYPE html>
<html>
<body>
@{Html.BeginForm("Browse", "Home");}
@Html.Label("User Name:")
@Html.TextBox("textBox1")
@Html.Label("Address:")
@Html.TextArea("textareal")
@Html.Label("Password:")
@Html.Password("password")
@Html.Label("I need updates on my mail:")
@Html.CheckBox("checkbox1")
@Html.Label("Select your city:")
@Html.DropDownList("myList", new SelectList(new [] {"New York", "Philadelphia", "California"}), "Choose")
Male @Html.RadioButton("Gender", "Male", true)
Female @Html.RadioButton("Gender", "Female")
<input type="submit" value="Submit">
@{Html.EndForm();}
</body>
</html>
```

Code Snippet 34 uses two `Html.RadioButton()` helper methods to create two radio buttons to accept the gender of the user.

Figure 3.27 shows the output of the `Html.RadioButton()` helper method.

The screenshot shows a Microsoft Edge browser window with the URL `http://localhost:298`. The page displays a user registration form. The fields include:

- User Name: An input text field.
- Address: A multi-line text area with scroll bars.
- Password: An input text field.
- I need updates on my mail: A checkbox input field, which is unchecked.
- Select your city: A dropdown menu currently set to "Choose".
- Male: A radio button input field, which is checked.
- Female: A radio button input field, which is unchecked.
- Submit: A grey rectangular button at the bottom of the form.

Figure 3.27: Using the `Html.RadioButton()` Helper Method

3.3.10 Using `Url.Action()`

You can use the `Url.Action()` helper method to generate a URL that targets a specified action method of a controller.

Syntax:

```
@Url.Action(<action_name>, <controller_name>)
```

where,

`action_name`: Is the name of the action method.

`ControllerName`: Is the name of the controller class.

Code Snippet 35 shows use of the `Url.Action()` method.

Code Snippet 35:

```
<!DOCTYPE html>
<html>
<body>
<a href='@Url.Action("Browse", "Home")'>Browse</a>
</body>
</html>
```

The code creates a hyperlink that targets the URL generated using the `Url.Action()` method. When the user clicks the hyperlink, the `Browse` action of the `Home` controller will be invoked.

Figure 3.28 shows the output of the `Url.Action()` method.

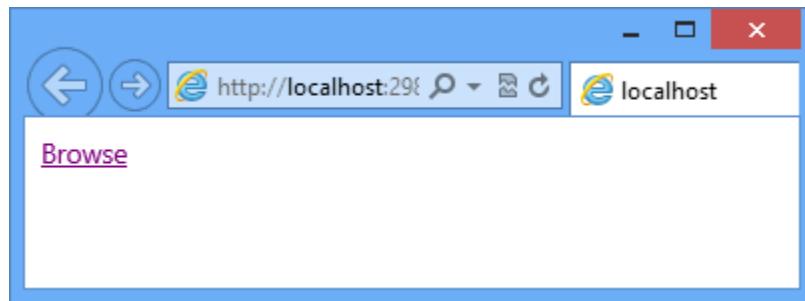


Figure 3.28: Using the `Url.Action()` Method

3.4 Check Your Progress

1. Which of these statements about Views are false?

(A)	Views are used to display form fields to accept user inputs.		
(B)	Views are used to display content of an application.		
(C)	View automatically configures a route in the <code>RouteConfig.cs</code> file, when you create an ASP.NET MVC application.		
(D)	Views in an ASP.NET MVC application provide the UI of the application to the user.		
(E)	View allows you to create a route constraint using regular expression.		

(A)	A, B	(C)	C, E
(B)	B, C	(D)	C, D

2. Which of these statements about `ViewData` are true?

(A)	Value of <code>ViewData</code> becomes null if the request is redirected.		
(B)	<code>ViewData</code> allows you to pass data from a model to a view.		
(C)	<code>ViewData</code> allows you to create multiple action methods in a controller.		
(D)	<code>ViewData</code> allows you to specify values by using key-value pairs in the action method.		
(E)	<code>ViewData</code> is a dynamic property base on the dynamic features introduced in C# 4.0.		

(A)	A, E	(C)	B, C
(B)	A, D	(D)	B, E

3. Which of these statements about Razor syntax rules are true and which statements are false?

(A)	Enclose code blocks between <code>@{</code> and <code>}</code>		
(B)	Start inline expressions with <code>%</code>		
(C)	End a Razor code statement with semicolon <code>(;)</code>		
(D)	Enclose strings in quotation marks		
(E)	Declare variables using any keyword		

(A)	A, C	(C)	D, E
(B)	C, D	(D)	B, E

4. In an ASP.NET MVC application, you need to render a view, named `Welcome.cshtml` that is present in a different folder, named `/Views/Demo` folder, instead of the default folder. Which of the following code will help you to achieve this?

(A)	<pre>public class HomeController : Controller { public ActionResult Index() { return View("/Views/Demo/Welcome.cshtml"); } }</pre>
(B)	<pre>public class HomeController : Controller { public ActionResult Index() { return View("~/Views/Home/Demo/Welcome.cshtml"); } }</pre>
(C)	<pre>public class HomeController : Controller { public ActionResult Index() { return View("~/Welcome.cshtml/Views/Demo"); } }</pre>
(D)	<pre>public class HomeController : Controller { public ActionResult Index() { return View("~/Views/Demo/Welcome.cshtml"); } }</pre>

(A)	D	(C)	A
(B)	B	(D)	C

5. Assuming that you have the following controller action that stores a ViewBag property.

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.CommonMessage = "Message to access";
        return View();
    }
}
```

Now, you need to access both ViewData and ViewBag to access the CommonMessage property stored in ViewBag. Which of the following code will help you to achieve this?

(A)	<!DOCTYPE html> <html> <body> <p> From ViewData: @ViewData["CommonMessage"] </p> <p> From ViewBag: @ViewBag["CommonMessage"] </p> </body> </html>
(B)	<!DOCTYPE html> <html> <body> <p>

(B)	<pre>From ViewData: @ViewData["CommonMessage"] </p> <p> From ViewBag: @ViewBag.CommonMessage </p> </body> </html></pre>
(C)	<pre><!DOCTYPE html> <html> <body> <p> From ViewData: @ViewData.CommonMessage </p> <p> From ViewBag: @ViewBag["CommonMessage"] </p> </body> </html></pre>
(D)	<pre><!DOCTYPE html><html> <body> <p> From ViewData: @ViewData.CommonMessage </p> <p> From ViewBag: @ViewBag.CommonMessage </p> </body> </html></pre>

(A)	C	(C)	B
(B)	A	(D)	C

3.4.1 Answers

(1)	C
(2)	B
(3)	D
(4)	A
(5)	C



Summary

- In an ASP.NET MVC application, views are used to display both static and dynamic content.
- View engines are part of the MVC Framework that converts the code of a view into HTML markup that a browser can understand.
- You can use ViewData, ViewBag, and TempData to pass data from a controller to a view.
- In an ASP.NET MVC application, a partial view represents a sub-view of a main view.
- Razor is the syntax, based on the ASP.NET Framework that allows you to create views.
- The MVC Framework uses a view engine to convert the code of a view into HTML markup that a browser can understand.
- The MVC Framework provides built-in HTML helper methods that you can use to generate HTML markup and you can reuse it across the Web application.



Balanced Learner-Oriented Guide

for enriched learning available



www.onlinevarsity.com

Session - 4

Models in ASP.NET MVC

Welcome to the Session, **Models in ASP.NET MVC**.

In an ASP.NET MVC application, the model component represents data associated with the application. The MVC Framework provides a model binder that is responsible for binding requests to model properties. Visual Studio.NET provides scaffolding facility that allows automatic creation of views based on model properties.

In this Session, you will learn to:

- ➔ Define and describe models
- ➔ Explain how to create a model
- ➔ Describe how to pass model data from controllers to views
- ➔ Explain how to create strongly typed models
- ➔ Explain the role of the model binder
- ➔ Explain how to use scaffolding in Visual Studio.NET

4.1 Introducing Models

In an ASP.NET MVC application, a model represents data associated with the application. Consider a scenario where you need to create a registration module of an online shopping store implemented using ASP.NET MVC. This module enables a user to register with the application. In the registration module, you will need user information, such as user id, user name, address, and e-mail id that you will store in a database. You will also need to display the user information when the user requests for them or when they want to update the information. In an ASP.NET MVC application, you represent such information as a model. A model is a class containing properties that represents data of the application.

4.1.1 Types of Model

The ASP.NET MVC Framework is based on the MVC pattern. This pattern defines the following three types of models, where each model has specific purpose.

The three types of model are as follows:

- **Data model:** Represent classes that interact with a database. Data models are set of classes that can either follow the database-first approach or code-first approach.
- **Business model:** Represent classes that implement a functionality that represents business logic of an application. While processing the business logic of an application, the classes of this model can interact with the classes included in the data model to retrieve and save data in the database.
- **View model:** Represent classes that provide information passed between controllers and views. Thus, the view identifies what to display in the browser. The classes of the view model do not process anything, instead it contain only the data for the view to display properly.

Note - Out of the three types of model, the view model is the most relevant model for developing ASP.NET MVC application.

4.1.2 Creating a Model

To create a model in an ASP.NET MVC application, you need to create a public class. In this model class, you need to declare public properties for each information that the model represents.

Code Snippet 1 shows declaring a model class named, **User**.

Code Snippet 1:

```
public class User
{
    public long Id { get; set; }
    public string name { get; set; }
    public string address { get; set; }
```

```
public string email { get; set; }
}
```

This code creates a model class named, `User` that contains the `Id`, `name`, `address`, and `email` properties declared as `public`.

Note - By convention, you should store model classes in the **Models** folder.

4.1.3 Accessing a Model within a Controller

In an ASP.NET MVC application, when the user request for some information, the request is received by an action method. In the action method, you need to access the model that stores the data.

To access the model in the action method, you need to create an object of the model class and either retrieve or set the property values of the object.

Code Snippet 2 shows creating an object of the model class in the `Index()` action method.

Code Snippet 2:

```
public ActionResult Index()
{
    var user = new MVCModelDemo.Models.User();
    user.name = "John Smith";
    user.address = "Park Street";
    user.email = "john@mvceexample.com";
    return View();
}
```

In this code, `user` is an object of the `User` class. The property values of the model is set to the data related to a user, such as name, address, and e-mail id.

4.1.4 Passing Model Data from a Controller to a View

Once you have accessed the model within a controller, you need to make the model data accessible to a view, so that the view can display the data to the user. To achieve this, you can pass the model object to the view, while invoking the view. You can also pass a collection of model objects from a controller to a view.

→ **Passing a Single Object**

In an action method, you can create a model object and then, pass the object to a view by using the `ViewBag` object.

Code Snippet 3 shows passing the `User` model data from an action method to a view by using a `ViewBag` object.

Code Snippet 3:

```
Public ActionResult Index()
{
    var user = new MVCModelDemo.Models.User();
    user.name = "John Smith";
    user.address = "Park Street";
    user.email = "john@mvceexample.com";
    ViewBag.user = user;
    return View();
}
```

In this code, an object of the `User` model class is created and initialized with values. The object is then, passed to the view by using a `ViewBag` object.

Now, you can access the data of the model object stored in the `ViewBag` object from within the view.

Code Snippet 4 shows accessing the data of the model object stored in the `ViewBag` object.

Code Snippet 4:

```
<!DOCTYPE html>
<html>
<body>
<p>
    User Name: @ViewBag.user.name
</p>
<p>
    Address: @ViewBag.user.address
</p>
<p>
    Email: @ViewBag.user.email
</p>
</body>
</html>
```

In this code, the view accesses and displays the name, address, and email properties of the User model object stored in the ViewBag object.

→ **Passing a Collection of Model Objects**

Consider a scenario where you need to display the details of multiple users. For this, you need to create and initialize multiple model objects and then, pass those objects to the view. You can pass multiple model objects to the view by adding the objects to a collection and passing the collection to the view.

Code Snippet 5 shows passing a collection of model objects to a view.

Code Snippet 5:

```
public ActionResult Index()
{
    var user = newList<User>();
    var user1 = new User();
    user1.name = "Mark Smith";
    user1.address = "Park Street";
    user1.email = "Mark@mvceexample.com";
    var user2 = new User();
    user2.name = "John Parker";
    user2.address = "New Park";
    user2.email = "John@mvceexample.com";
    var user3 = new User();
    user3.name = "Steave Edward";
    user3.address = "Melbourne Street";
    user3.email = "steave@mvceexample.com";
    user.Add(user1);
    user.Add(user2);
    user.Add(user3);

    ViewBag.user = user;
    return View();
}
```

This code creates and initializes three objects of the model class, named `User`. Then, a `List<User>` collection is created and the model objects are added to it. Finally, the collection is passed to the view by using a `ViewBag` object.

Once you pass a collection of model objects to a view using a `ViewBag` object, you can retrieve the collection stored in the `ViewBag` object from within the view, iterate through the collection to retrieve each model object, and access their properties.

Code Snippet 6 shows retrieving model objects from a collection and displaying their properties.

Code Snippet 6:

```
<!DOCTYPE html>
<html>
<body>
<h3>User Details</h3>
@{
    var user = ViewBag.user;
}
@foreach (var p in user)
{
    @p.name<br />
    @p.address<br />
    @p.email<br />
<br/>
}
</body>
</html>
```

This code uses a `foreach` loop to iterate through the collection of model objects stored in the `ViewBag` object. Then, for each model object, the `name`, `address`, and `email` properties are rendered as a response.

Figure 4.1 shows the output of retrieving model objects.

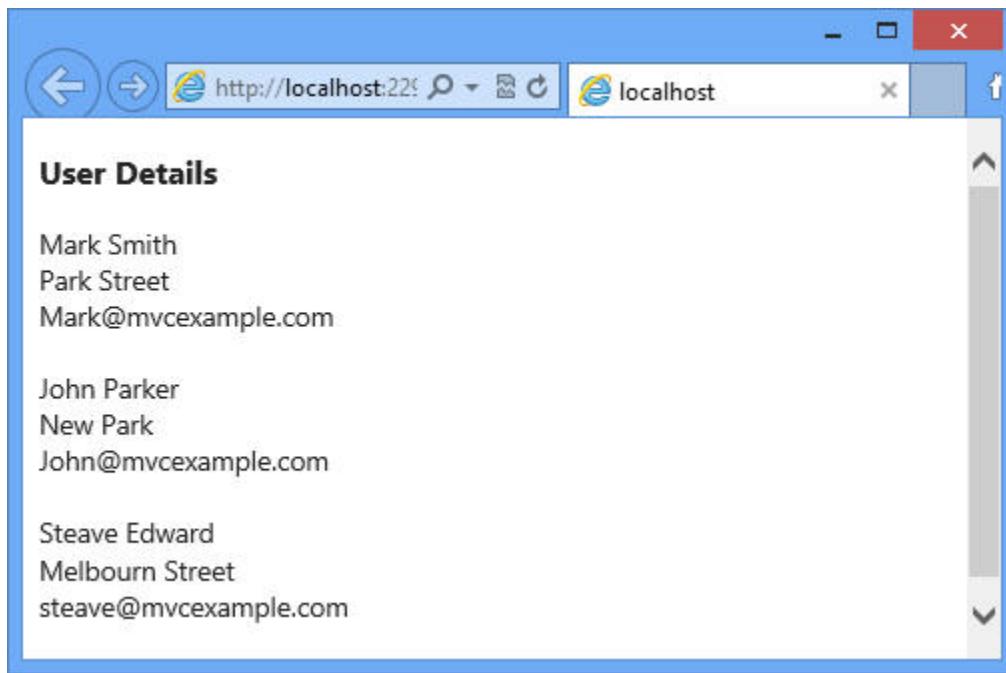


Figure 4.1: Output of Retrieving Model Objects

You have learned how to pass a collection of model objects from an action method to a view using a `ViewBag` object. Another approach to pass a collection of model objects from an action method to a view is to pass the collection directly as a parameter to the `View()` method.

Code Snippet 7 shows passing a collection of model objects to a view as a parameter to the `View()` method.

Code Snippet 7:

```
public ActionResult Index()
{
    var user = new List<User>();
    var user1 = new User();
    user1.name = "Mark Smith";
    user1.address = "Park Street";
    user1.email = "Mark@mvceexample.com";
    var user2 = new User();
    user2.name = "John Parker";
    user2.address = "New Park";
    user2.email = "John@mvceexample.com";
}
```

```

var user3 = new User();
    user3.name = "Steave Edward ";
    user3.address = "Melbourne Street";
    user3.email = "steave@mvceexample.com";
user.Add(user1);
user.Add(user2);
user.Add(user3);
return View(user);
}

```

This code creates and initializes three objects of the models class, named `User`. Then, a `List<User>` collection is created and the model objects are added to it. Finally, the collection is passed to the view as a parameter to the `View()` method.

Code Snippet 8 shows retrieving the user information in the view.

Code Snippet 8:

```

<!DOCTYPE html>
<html>
<body>
<h3>User Details</h3>
@{
    var user=Model;
}
@foreach(var p in user)
{
    @p.name<br />
    @p.address<br />
    @p.email<br />
<br/>
}
</body>
</html>

```

This code shows how to retrieve the user information that has been passed to a view by passing a collection of objects as a parameter.

4.1.5 Using Strong Typing

While passing model data from a controller to a view, the view cannot identify the exact type of the data. As a result, you face problem when trying to access the properties of the passed object. In addition, you are also deprived of the benefits of strong typing and compile-time checking of code.

As a solution, you can typecast the model data to a specific type.

Code Snippet 9 shows how to typecast model data.

Code Snippet 9:

```
<html>
<body>
<h3>User Details</h3>
@{
    var user = Model as MVCModelDemo.Models.User;
}
@user.name<br/>
@user.address<br/>
@user.email<br/>
</body>
</html>
```

In this code, the `Model` object is cast to the type, `MVCModelDemo.Models.User`. As a result of this casting, the `user` object is created as an object of the type, `MVCModelDemo.Models.User`, and enables compile-time checking of code.

You can also ignore explicit type casting of a model object, by creating a strongly typed view. A strongly typed view specifies the type of a model it requires by using the `@model` keyword.

Syntax:

`@model <model_name>`

where,

`model_name`: Is the fully qualified name of the model class.

Once you use the `@model` keyword, you can access the properties of the model object in the view.

Code Snippet 10 shows accessing the properties of the model object by using the `@model` keyword.

Code Snippet 10:

```
@model MVCModelDemo.Models.User
<html>
```

```
<body>
<h3>User Details</h3>
@Model.name<br/>
@Model.address<br/>
@Model.email<br/>
</body>
</html>
```

Figure 4.2 shows the output of accessing properties of the model object.

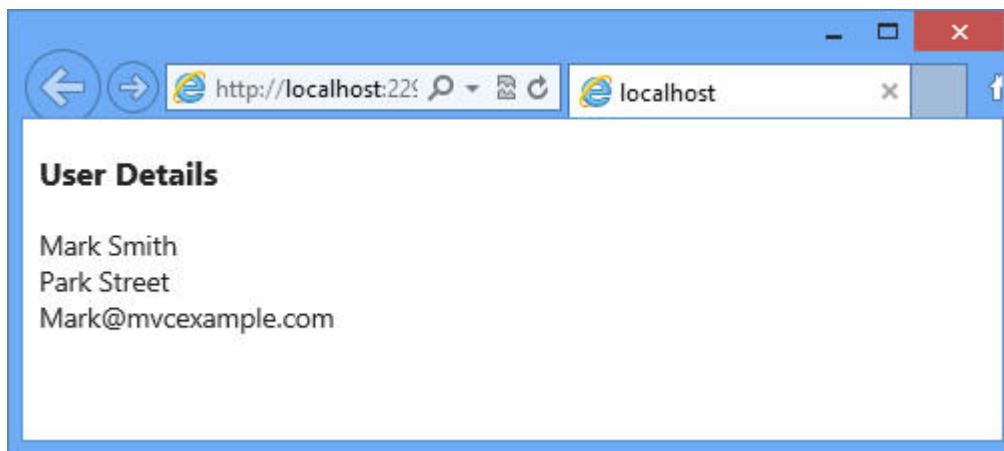


Figure 4.2: Output of Accessing Properties of Model Object

Sometime, you may need to pass a collection of objects to a view. In such situation, you can use the @model keyword.

Code Snippet 11 shows using the @model keyword to pass a collection of model object.

Code Snippet 11:

```
@model IEnumerable<MVCMODELDEMO.Models.User>
```

This code uses the @model keyword to indicate that it expects a collection of the User model objects.

Once you pass a collection of the model objects, you can access it in a view.

Code Snippet 12 shows accessing the collection of the User model in a view.

Code Snippet 12:

```
@model IEnumerable<MVCMODELDEMO.Models.User>
<html>
<body>
<h3>User Details</h3>
```

```

@{
var user=Model;
}

@foreach (var u in user)
{
    @u.name<br/>
    @u.address<br/>
    @u.email<br/>
<br/>
}
</body>
</html>

```

Figure 4.3 shows the output of accessing collection of the User model.

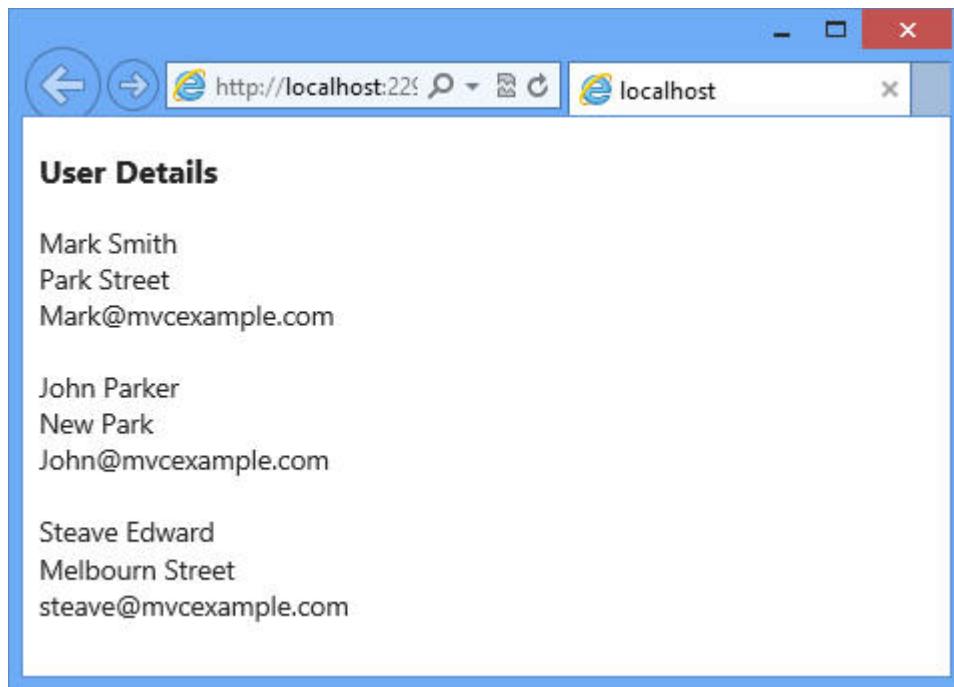


Figure 4.3: Output of Accessing Collection of User Model

4.1.6 HTML Helper Methods in Strongly Types Views

You have learned about HTML helper methods that you can use in views to generate HTML markup for UI elements, such as forms, labels, text fields, text areas, and radio buttons. The MVC Framework enables these helper methods to directly associate with model properties in a strongly typed views. In addition, the MVC Framework provides helper methods that you can use only in strongly typed views.

Table 4.1 lists the helper methods that you can use only in strongly typed views.

Helper Method	Description
Html.LabelFor()	Is the strongly typed version of the Html.Label() helper method. This method uses a lambda expression as its parameter, which provides compile time checking.
Html.DisplayNameFor()	Is used to display the names of model properties.
Html.DisplayFor()	Is used to display the values of the model properties.
Html.TextBoxFor()	Is the strongly typed version of the Html.TextBox() helper method.
Html.TextAreaFor()	Is the strongly typed version of the Html.TextArea() helper method. This method generates the same markup as that of the Html.TextArea() helper method.
Html.EditorFor()	Is used to display an editor for the specified model property.
Html.PasswordFor()	Is the strongly typed version of the Html.Password() helper method. It is used to render a password field.
Html.CheckBoxFor()	Is the strongly typed version of the Html.CheckBox() helper method. It renders a check box input element that enables the user to select a true or false condition.
Html.DropDownListFor()	Is the strongly typed version of the Html.DropDownList() helper method that allows selection of a single item.
Html.RadioButtonFor()	Is the strongly typed version of the Html.RadioButton() helper method. This method takes an expression that identifies the object that contains the property to render, followed by a value to submit when the user selects the radio button.

Table 4.1: Helper Methods for Strongly Typed Views

Code Snippet 13 shows using HTML helper methods in a strongly typed view.

Code Snippet 13:

```
@model MVCModelDemo.Models.User
@{
    ViewBag.Title = "User Form";
}
<h2>User Form</h2>
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
<div>
```

```
@Html.LabelFor(model => model.name)  
</div>  
  
<div>  
@Html.EditorFor(model => model.name)  
</div>  
  
<div>  
@Html.LabelFor(model => model.address)  
</div>  
  
@Html.EditorFor(model => model.address)  
<div>  
@Html.LabelFor(model => model.email)  
</div>  
  
<div>  
@Html.EditorFor(model => model.email)  
</div>  
  
<p>  
<input type="submit" value="Create" />  
</p>  
}
```

In this code, the `Html.LabelFor()` method is used to display labels based on the property names of the model. The `Html.EditorFor()` method is used to display editable fields for the properties of the model.

Figure 4.4 shows the output of using HTML helper methods.

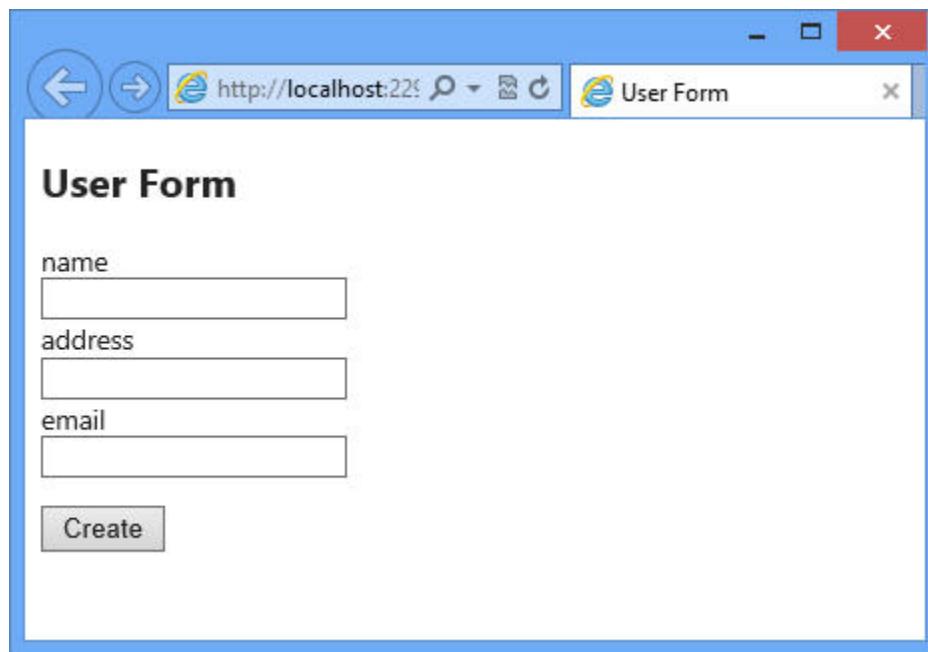


Figure 4.4: Output of Using HTML Helper Methods

4.2 Role of Model Binder

You have learned that you can display various input fields in a view, such as labels, text boxes, check boxes, radio buttons, and so on. When the user submits values using these fields in a form, the user specified information is sent to the controller.

When a user submits information in a form within a strongly typed view, ASP.NET MVC automatically examines the `HttpRequest` object and maps the information sent to fields in the model object. The process of mapping the information in the `HttpRequest` object to the model object is known as model binding.

There are several advantages of model binding. Some of them are as follows:

- Automatically extract the data from the `HttpRequest` object.
- Automatically converts data type.
- Makes data validation easy.

The MVC Framework provides a model binder that performs model binding in application. The `DefaultModelBinder` class implements the model binder on the MVC Framework. The two most important roles of the model binder are as follows:

- Bind request to primitive values

- Bind request to objects

4.2.1 Binding to Primitive Values

To understand how the model binder binds request to primitive values, consider a scenario where you are creating a login form that accepts login details from user.

For this, first you need to create a `Login` model in your application.

Code Snippet 14 shows the `Login` model class.

Code Snippet 14:

```
public class Login
{
    public string userName { get; set; }

    [DataType(DataType.Password)]
    public string password { get; set; }
}
```

This code creates two properties named, `userName` and `password` in the `Login` model.

After creating the model class, you need to create an `Index.cshtml` view to display the login form.

Code Snippet 15 shows the content of the `Index.cshtml` file.

Code Snippet 15:

```
@model ModelDemo.Models.Login
@{
    ViewBag.Title = "Index";
}

<h2>User Details</h2>
@using (Html.BeginForm())
{
    @Html.ValidationSummary(true)
    <div>
        @Html.LabelFor(model => model.userName)
    </div>
    <div>
        @Html.EditorFor(model => model.userName)
```

```

</div>
<div>
@Html.LabelFor(model => model.password)
</div>
<div>
@Html.EditorFor(model => model.password)
</div>
<div>
<input type="submit" value="Submit" />
</div>
}

```

This code uses the `Html.EditorFor()` method to display the UI fields for the `username` and `password` properties of the `Login` model.

Once you have created the view, you need to create a controller class that contains the `Index()` action method to display the view. You also need to create another `Index()` action method with the `HttpPost` attribute that will receive the user submitted login data.

Code Snippet 16 shows the `HomeController` controller.

Code Snippet 16:

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Index(string userName, string password)
    {
        if (userName == "Peter" && password == "pass@123")
        {
            string msg = "Welcome" + userName;
            return Content(msg);
        }
    }
}

```

```
    else
    {
        return View();
    }
}
```

In this code, the first `Index()` method returns the `Index.cshtml` view that displays a login form. The second `Index()` method is marked with the `HttpPost` attribute. This method accepts two primitive as parameters. The `Index()` method compares the parameters with predefined values and returns a message if the comparison returns true. Else, the `Index()` method returns back the `Index.cshtml` view.

When a user submits the login data, the default model binder maps the values of the `userName` and `password` fields to the primitive type parameters of the `Index()` action method. In the `Index()` action method, you can perform the required authentication and return a result.

4.2.2 Binding to Object

To understand how the model binder binds requests to objects, consider the same scenario where you are creating a login form. For this, you have already created the `Login` model and the `Index.cshtml` view.

Now, to bind request to object, you need to update the controller class so that it accepts a `Login` object as a parameter, instead of an `HttpRequest` object.

Code Snippet 17 shows the updated controller class.

Code Snippet 17:

```
public class HomeController : Controller
{
    public ActionResult Index() {
        return View();
    }

    [HttpPost]
    public ActionResult Index(Login login)
    {
        if (login.userName == "Peter" && login.password == "pass@123")
        {
            // ...
        }
    }
}
```

```
string msg = "Welcome " + login.userName;

return Content(msg);

}

else

{

    return View();

}

}
```

In this code, the first `Index()` method returns the `Index.cshtml` view that displays a login form. The second `Index()` method automatically removes the data from the `HttpRequest` object and put into the `Login` object.

When a user submits the login data, the `Index()` method validates the username and password passed in the `Login` object. When the validation is successful, the view displays a welcome message.

When you access the application from the browser, the `Index.cshtml` view displays the login form.

1. Type Peter in the User Name text field and pass@123 in the Password field of the login form.

Figure 4.5 shows the login form with data the specified in the **User Name** and **Password** fields.

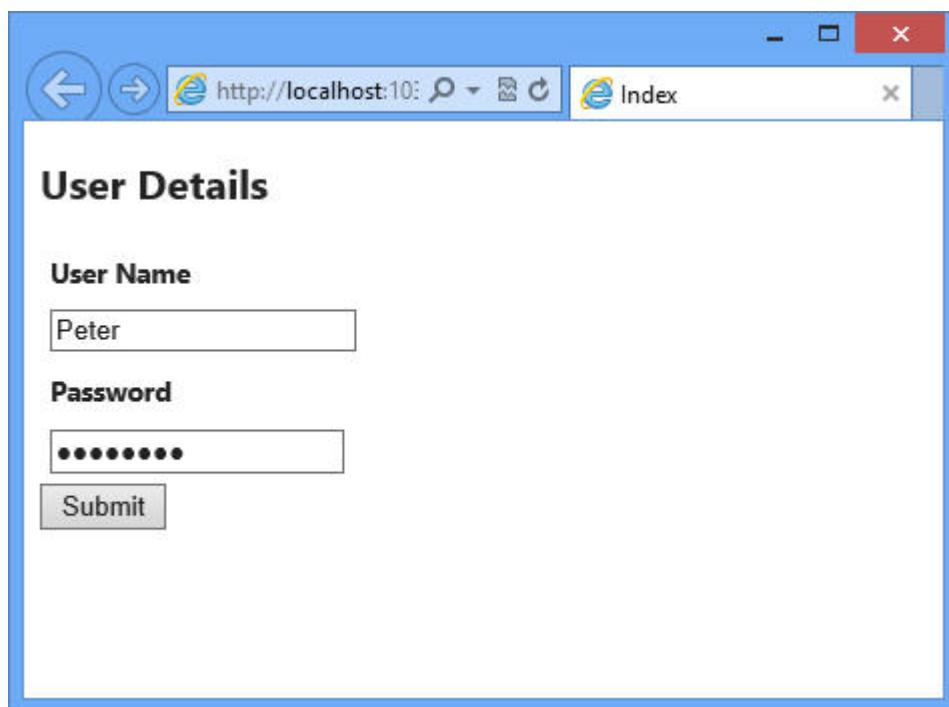


Figure 4.5: Login Form

- Click **Submit**. The login form displays a Welcome Peter message.

Figure 4.6 shows the output of the `Index.cshtml` view after submitting the user name and password.

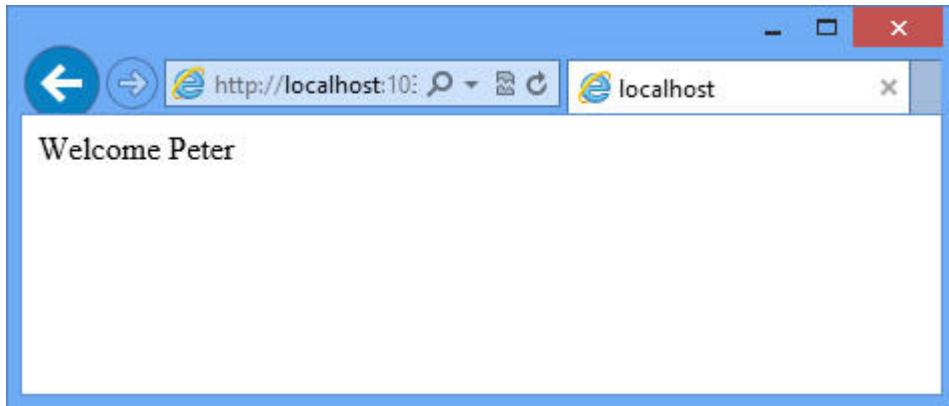


Figure 4.6: Output of `Index.cshtml` View

4.3 Visual Studio .NET Scaffolding

While developing an application in Visual Studio .NET, you can create a view either manually or automatically based on the model property. The ASP.NET MVC Framework provides a feature called scaffolding that allows you to generate views automatically.

By convention, scaffolding uses specific name for views. After creating a view it stores the auto-generated code in respective places for the application to work.

There are five types of template that the scaffolding feature provides to create views. They are as follows:

- **List**: Generates markup to display the list of model objects.
- **Create**: Generates markup to add a new object to the list.
- **Edit**: Generates markup to edit an existing model object.
- **Details**: Generates markup to show the information of an existing model object.
- **Delete**: Generates markup to delete an existing model object.

4.3.1 List Template

You can use the List template to create a view that displays a list of model objects. This list of model object is passed to a view by using an action method.

Code Snippet 18 shows an `Index()` action method that returns an `ActionResult` object through a call

to the `View()` method of the controller class.

Code Snippet 18:

```
public ActionResult Index()
{
    var user = new List<User>();
    //Code to populate the user collection
    return View(user);
}
```

This code shows the `Index()` action method of a controller that returns the result of a call to the `View()` method. The result of the `View()` method is an `ActionResult` object that renders a view.

Visual Studio .NET simplifies the process of creating a view for an action method using the List scaffolding template. To create a view using the List template, you need to perform the following steps:

1. Right-click inside the action method for which you need to create a view.
2. Select **Add View** from the context menu that appears. The **Add View** dialog box appears.

Figure 4.7 shows the **Add View** dialog box.

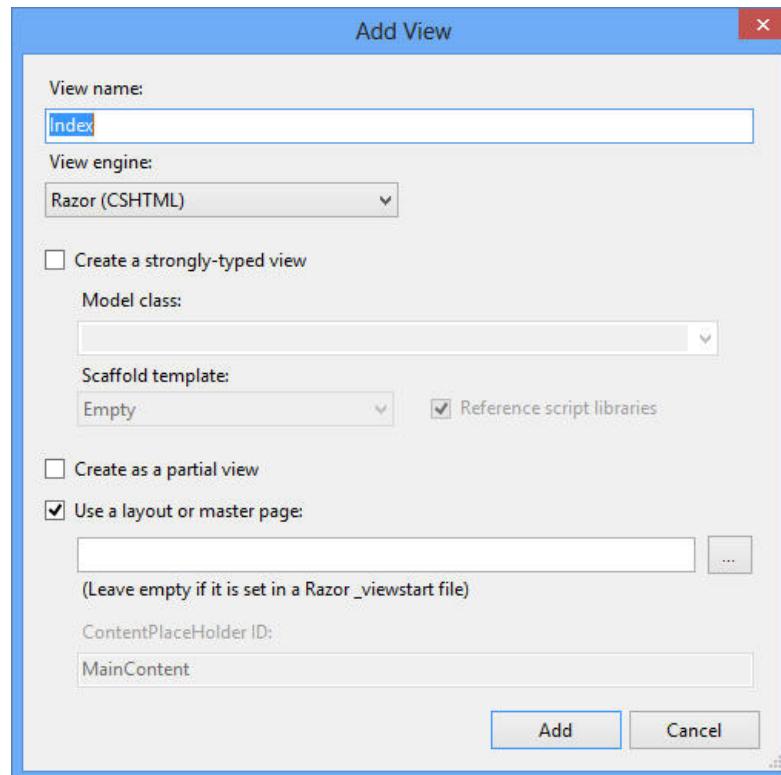


Figure 4.7: Add View Dialog Box

3. Select the **Create a strongly-typed view** check box. This will enable the fields that allow you to specify the model class and scaffolding template.
4. Select the model class from the **Model class** drop-down list.
5. Select **List** from the **Scaffold template** drop-down list.

Figure 4.8 shows specifying the model class and scaffolding template in the **Add View** dialog box.

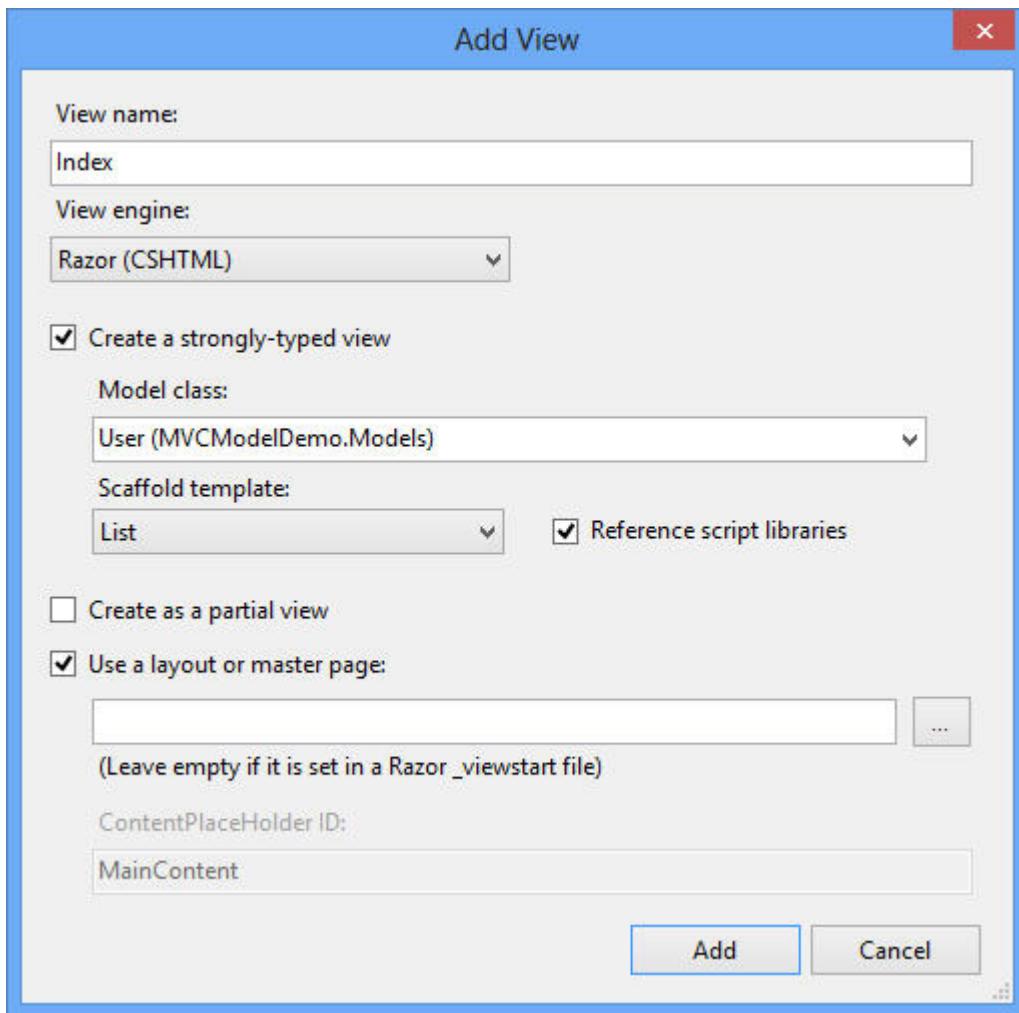


Figure 4.8: Selecting Scaffold Template

6. Click **Add**. Visual Studio .NET automatically creates the appropriate directory structure and adds the view file to it.

Code Snippet 19 shows the auto-generated markup when you create the view using the List template for the User model.

Code Snippet 19:

```
@model IEnumerable<MVCMODELDEMO.Models.User>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>

<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.address)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.email)
        </th>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.address)
            </td>
```

```
</td>
<td>
@Html.DisplayFor(modelItem => item.email)
</td>
<td>
@Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
@Html.ActionLink("Details", "Details", new { id=item.Id }) |
@Html.ActionLink("Delete", "Delete", new { id=item.Id })
</td>
</tr>
}
</table>
```

In this code,

- The `Html.DisplayNameFor()` helper method displays the names of model properties.
- The `Html.DisplayFor()` helper method displays the values of the model properties.
- The `Html.ActionLink()` helper method create links to edit, delete, and view details for a user.
- The `@model IEnumerable<MVCDemo.Models.User>` method is used to type the view to a collection of User objects.
- The `@foreach(var item in Model)`syntax iterates over the collection of users and displays the property values associated with each user individually.
- The `Html.DisplayNameFor()` and `Html.DisplayFor()` helper methods use a parameter to specify the names and the values of the properties to be displayed on the view.

Figure 4.9 shows the output of the auto-generated markup.

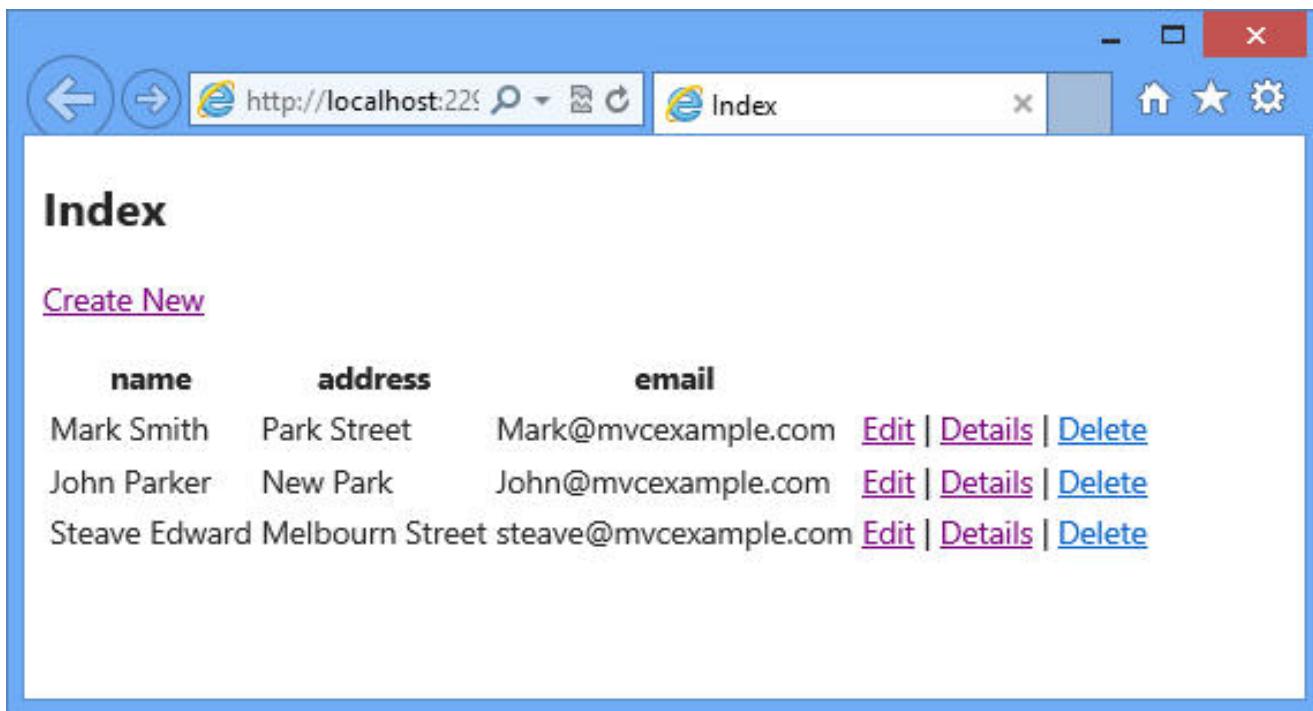


Figure 4.9: Output of Auto-generated Markup

4.3.2 Create Template

You can use the Create template to generate a view that accepts the details of a new object to be stored in a data store. You need to create an action method, to display a view based on the Create template.

Code Snippet 20 creates an action method named, `Create()` in the `HomeController` controller.

Code Snippet 20:

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

This code creates the `Create()` action method that will invoke, when a user clicks the `Create` link on the view generated using the List template.

For this, you need to first create a view using the Create template. To create a view using the Create template, you need to perform the following steps:

1. Right-click inside the `Create()` action method.
2. Select **Add View** from the context menu that appears. The **Add View** dialog box appears.
3. Select the **Create a strongly-typed view** checkbox.

4. Select the model class from the **Model class** drop-down list.
5. Select **Create from the Scaffold templates** drop-down list.
6. Click **Add**. Visual Studio .NET automatically creates a view named, Create in the appropriate directory structure.

Code Snippet 21 shows the auto-generated markup when you create the view using the Create template.

Code Snippet 21:

```
@model MVCModelDemo.Models.User  
{@  
    ViewBag.Title = "Create";  
}  


## Create

<@using(Html.BeginForm()) {  
    @Html.ValidationSummary(true)  
    <fieldset>  
        <legend>User</legend>  
        <div class="editor-label">  
            @Html.LabelFor(model => model.name)  
        </div>  
        <div class="editor-field">  
            @Html.EditorFor(model => model.name)  
            @Html.ValidationMessageFor(model => model.name)  
        </div>  
        <div class="editor-label">  
            @Html.LabelFor(model => model.address)  
        </div>  
        <div class="editor-field">  
            @Html.EditorFor(model => model.address)  
            @Html.ValidationMessageFor(model => model.address)  
        </div>
```

```
<div class="editor-label">
    @Html.LabelFor(model => model.email)
</div>

<div class="editor-field">
    @Html.EditorFor(model => model.email)
    @Html.ValidationMessageFor(model => model.email)
</div>

<p>
    <input type="submit" value="Create" />
</p>
</fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

In this code,

- The `BeginForm` helper method starts a form.
- The `Html.ValidationSummary()` validation helper displays the summary of all the error messages at one place.
- The `Html.LabelFor()` helper method displays an HTML label element with the name of the property.
- The `Html.EditorFor()` helper method displays a textbox that accepts the value of a model property.
- The `Html.ValidationMessageFor()` validation helper method displays a validation error message for the associated model property.

Figure 4.10 shows the output of auto-generated markup using the Create template.

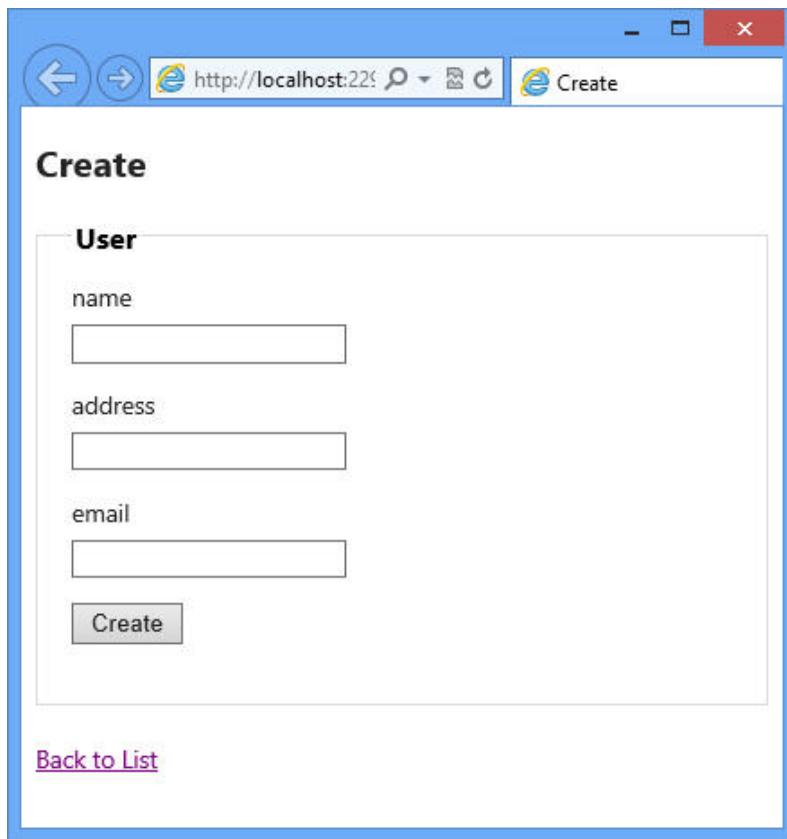


Figure 4.10: Output of Auto-generated Markup Using the Create Template

When the user enters data in the form and clicks the **Create** button, an HTTP POST request is sent to the `Create()` action method of the controller. In addition to that, the user entered data in the form will also be sent along with the request.

4.3.3 Edit Template

You can use the Edit template whenever you need to generate a view that required to be used for modifying the details of an existing object stored in a data store.

To display a view based on the Edit template, you need to create an action method to pass the model object to be edited to the view.

Code Snippet 22 shows the action method named, `Edit()`.

Code Snippet 22:

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

Once you have created the `Edit()` action method in the controller, you can use Visual Studio .NET to create the view using the Edit template. To create the view using the Edit template in Visual Studio .NET, you need to select **Edit** from the **Scaffold templates** drop-down list.

After creating a view named, `Edit` for the `User` model using the Edit template, Visual Studio .NET generates the markup for the view.

Code Snippet 23 shows the auto-generated markup when you create the view using the Edit template.

Code Snippet 23:

```
@model MVCModelDemo.Models.User  
{  
    ViewBag.Title = "Edit";  
}  


## Edit



```
@using (Html.BeginForm()) {
 @Html.ValidationSummary(true)
 <fieldset>
 <legend>User</legend>
 @Html.HiddenFor(model => model.Id)
 <div class="editor-label">
 @Html.LabelFor(model => model.name)
 </div>
 <div class="editor-field">
 @Html.EditorFor(model => model.name)
 @Html.ValidationMessageFor(model => model.name)
 </div>
 <div class="editor-label">
 @Html.LabelFor(model => model.address)
 </div>
 <div class="editor-field">
 @Html.EditorFor(model => model.address)
 @Html.ValidationMessageFor(model => model.address)
 </div>
```


```

```
<div class="editor-label">
@Html.LabelFor(model => model.email)
</div>

<div class="editor-field">
@Html.EditorFor(model => model.email)
@Html.ValidationMessageFor(model => model.email)
</div>

<p>
<input type="submit" value="Save" />
</p>
</fieldset>
}

<div>
@Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
@Scripts.Render("~/bundles/jqueryval")
}
```

In this code,

- The `Html.LabelFor()` helper method displays an HTML label element with the name of the property.
- The `Html.EditorFor()` helper method displays a textbox to accept the value of a model property.
- The `Html.ValidationMessageFor()` helper method displays a validation error message.

Figure 4.11 shows the output of creating the view using the Edit template.

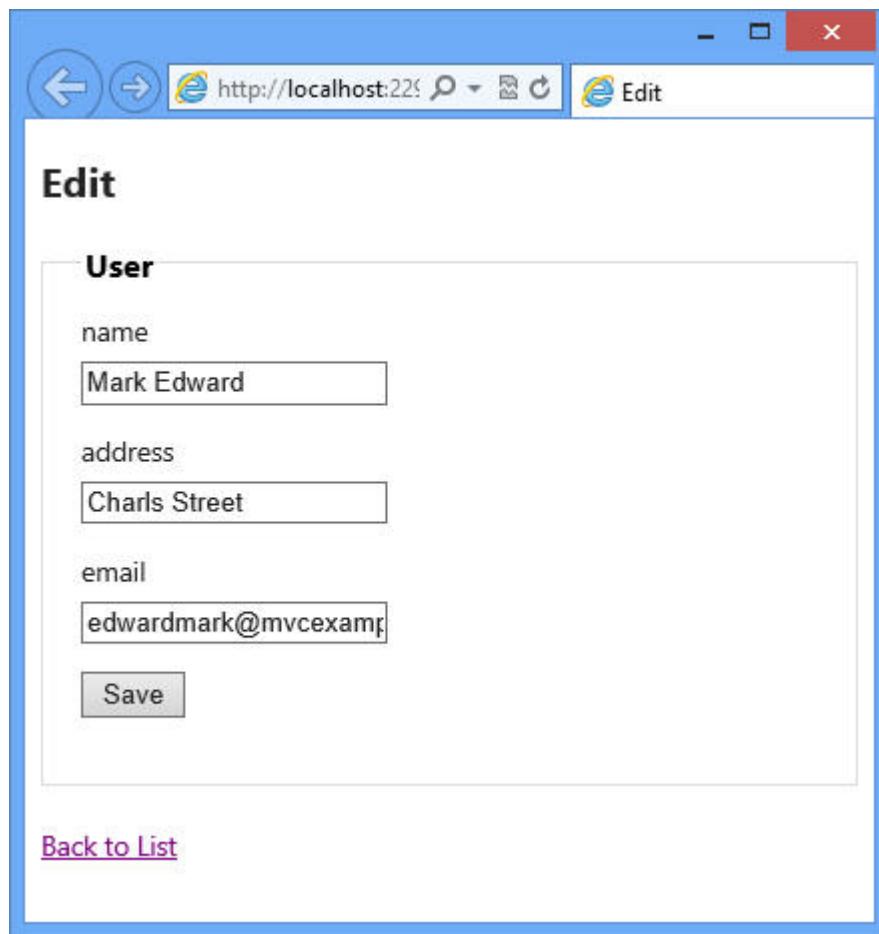


Figure 4.11: Output of Creating View Using the Edit Template

In this figure, the Edit view is displayed by the `Edit ()` action method of the `HomeController` controller. When the user edits data in the form and then, clicks the **Save** button, an HTTP POST request will be sent to the `Edit ()` action method. You can create another `Edit ()` action method in the controller to handle this HTTP POST request.

4.3.4 Details Template

You can use the Details template to create a view that displays details of the `User` model. The details of the `User` model object are passed to the view through an action method of the controller.

Once you have created the `Details ()` action method in the controller and a view using the Details template in Visual Studio .NET, it generates the markup for the view.

Code Snippet 24 shows the auto-generated markup when you create the view using the Details template.

Code Snippet 24:

```
@model MVCModelDemo.Models.User
@{
    ViewBag.Title = "Details";
}

<h2>Details</h2>
<fieldset>
    <legend>User</legend>
    <div class="display-label">
        @Html.DisplayNameFor(model => model.name)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.name)
    </div>
    <div class="display-label">
        @Html.DisplayNameFor(model => model.address)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.address)
    </div>
    <div class="display-label">
        @Html.DisplayNameFor(model => model.email)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.email)
    </div>
</fieldset>
<p>
    @Html.ActionLink("Edit", "Edit", new { id=Model.Id }) |

```

```
@Html.ActionLink("Back to List", "Index")
</p>
```

In this code, the `Html.DisplayNameFor()` helper method displays the name of model properties and the `Html.DisplayFor()` helper method displays the values of the model properties.

Figure 4.12 shows the output of Auto-generated markup for Details template.

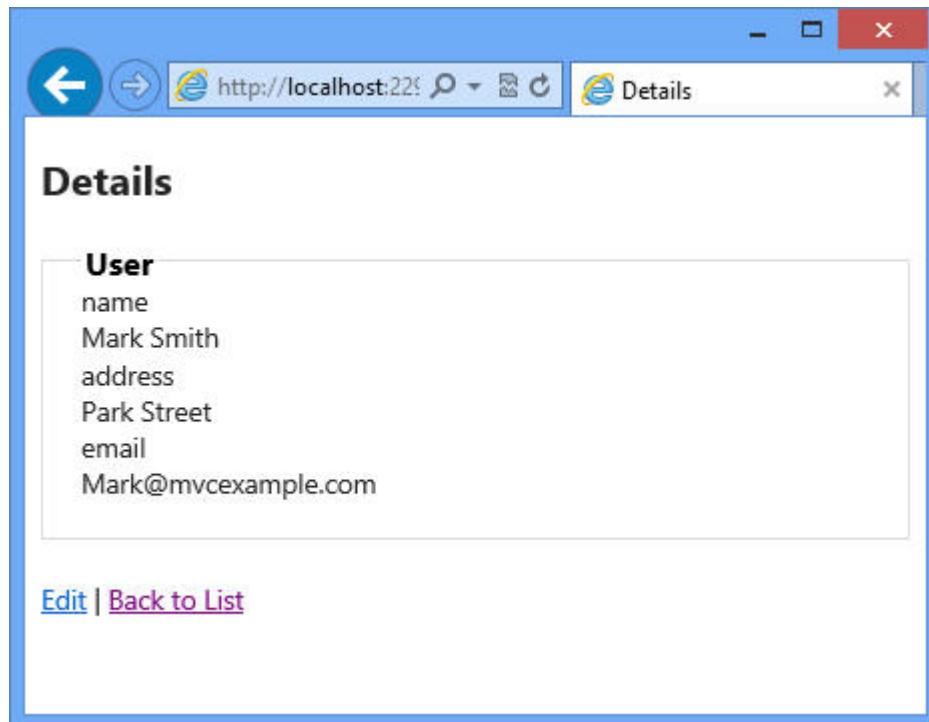


Figure 4.12: Output of Auto-generated Markup for Details Template

4.3.5 Delete Template

You can use the Delete template to generate a view that allows a user to delete an existing object from a data store. To create a view based on the Delete template, first you need to create an action method that passes the model object to be deleted to the view.

After you create the `Delete()` action method and a view for the `User` model using the Delete template in Visual Studio .NET, it generates the markup for the view.

Code Snippet 25 shows the auto-generated markup when you create a view using the Delete template.

Code Snippet 25:

```
@model MVCModelDemo.Models.User
@{
```

```
ViewBag.Title = "Delete";  
}  
  
<h2>Delete</h2>  
  
<h3>Are you sure you want to delete this?</h3>  
  
<fieldset>  
  
<legend>User</legend>  
  
<div class="display-label">  
    @Html.DisplayNameFor(model => model.name)  
</div>  
  
<div class="display-field">  
    @Html.DisplayFor(model => model.name)  
</div>  
  
<div class="display-label">  
    @Html.DisplayNameFor(model => model.address)  
</div>  
  
<div class="display-field">  
    @Html.DisplayFor(model => model.address)  
</div>  
  
<div class="display-label">  
    @Html.DisplayNameFor(model => model.email)  
</div>  
  
<div class="display-field">  
    @Html.DisplayFor(model => model.email)  
</div>  
  
</fieldset>  
  
@using (Html.BeginForm()) {  
  
    <p>  
        <input type="submit" value="Delete" /> |  
        @Html.ActionLink("Back to List", "Index")  
    </p>  
}
```

In this code, the `Html.DisplayNameFor()` helper method displays the names of model properties and the `Html.DisplayFor()` helper method displays the values of the model properties. In addition, the `Html.Actionlink()` helper method is used to create a link to list the details for a product.

Figure 4.13 shows the output of creating view using the Delete template.

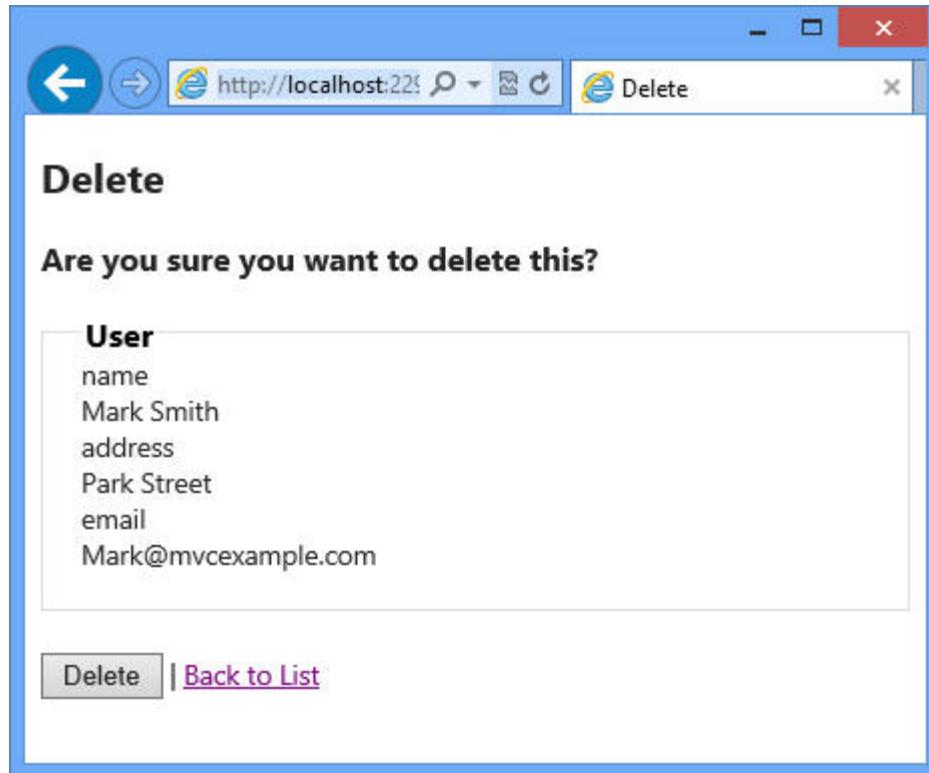


Figure 4.13: Output of Creating View Using the Delete Template

4.4 Check Your Progress

1. Which of the following statements about model are true?

(A)	A model manages the flow of the application in ASP.NET MVC application.
(B)	A model allows you to define URL patterns with place holders that maps to request URLs pattern.
(C)	A model allows you to create multiple action methods in a controller.
(D)	A model can contain classes that interact with a database.
(E)	A model is a class containing properties that provides data to the application.

(A)	A, B	(C)	A, D
(B)	C, D	(D)	E, D

2. Match the descriptions against their corresponding helper methods.

Helper Methods		Description	
(A)	Html.RadioButtonFor()	1.	This method displays an editor for the specified model property.
(B)	Html.DropDownListFor()	2.	This method displays the values of the model properties.
(C)	Html.EditorFor()	3.	This method takes an expression that identifies the object that contains the property to render, followed by a value to submit when the user selects.
(D)	Html.DisplayFor()	4.	This method allows selection of a single item.
(E)	Html.DisplayNameFor()	5.	This method displays the names of model properties.

(A)	A-3, B-4, C-1, D-2, E-5	(C)	A-3, B-4, C-5, D-1, E-2
(B)	A-2, B-1, C-3, D-4, E-5	(D)	A-5, B-4, C-1, D-3, E-2

3. Which of the following keyword a strongly typed view uses to specify the type of a model that it requires?

(A)	@model
(B)	@Model
(C)	%model%
(D)	_model
(E)	@model()

(A)	B	(C)	D
(B)	C	(D)	A

4. Which of the following code snippets will correctly access the collection of the `User` model in a strongly typed view?

Assuming that the fully qualified name of the model class is `ModelDemo.Models.User`.

(A)	<pre>@model IEnumerable<ModelDemo.Models.User> <html> <body> @{ var user = Model; } @foreach(var u in user) { @u.name
 @u.address
 @u.email
 }
 </body> </html></pre>
-----	---

(B)

```
@Model IEnumerable<ModelDemo.Models.User>

<html>
<body>
@{
    var user = Model;
}

@foreach(var u in user)
{
    @model.name <br/>
    @model.address<br/>
    @model.email<br/>
<br/>
}
</body>
</html>
```

(C)

```
%model ModelDemo.Models.User

<html>
<body>
@{
    var user = Model;
}

@foreach(var u in user)
{
    @u.name <br/>
    @u.address<br/>
    @u.email<br/>
<br/>
}
</body>
</html>
```

```
(D) @model IEnumerable<ModelDemo.Models>
<html>
<body>
    @{
        var user = Model;
    }
    @foreach(var u in user)
    {
        @u.name <br/>
        @u.address<br/>
        @u.email<br/>
        <br/>
    }
</body>
</html>
```

(A)	A	(C)	D
(B)	C	(D)	B

5. Which of the following statements are true about model binding?

(A)	Model binding is the process that maps incoming requests to specified controller actions.
(B)	Model binding is the process of mapping the information in the <code>HttpRequest</code> object to the model object.
(C)	Model binding is a process that allows binds requests only to primitive values.
(D)	Model binding is the process that allows rendering a partial view, which is a sub-view of the main view.
(E)	Model binding is a process that automatically extracts the data from the <code>HttpRequest</code> object.

(A)	A, B	(C)	B, C
(B)	C, D	(D)	B, E

4.4.1 Answers

(1)	C
(2)	A
(3)	D
(4)	A
(5)	D



Summary

- In an ASP.NET MVC application, a model represents data associated with the application.
- In the MVC pattern, there are three types of models, where each model has specific purpose.
- The MVC Framework provides helper methods that you can use only in strongly typed views.
- The process of mapping the data in an HttpRequest object to a model object is known as model binding.
- The MVC Framework provides a model binder that performs model binding in application.
- The ASP.NET MVC Framework provides a feature called scaffolding that allows you to generate views automatically.
- Visual Studio .NET simplifies the process of creating views for an action method using the different scaffolding template.

Session - 5

Data Validation and Annotation

Welcome to the Session, **Data Validation and Annotation**.

In an ASP.NET MVC application, you should ensure that any data submitted by a user to the application is valid. You can validate user input data by manually including validation logic in your application. Instead of manually validating user input data, you can use the validation workflow that the MVC Framework provides to validate user data. The MVC Framework provides several data annotations that you can apply as attributes to the properties of a model. These annotations can be used for several purposes, such as to validate user input data, to specify the UI field to display in a view for a property, and to specify that a property value is read-only.

The MVC Framework also provides the `ModelState` class that you can use in an application to check whether or not, the state of a model is valid.

In this Session, you will learn to:

- ➔ Define and describe how to validate data
- ➔ Explain how to use data annotation
- ➔ Explain and describe how to use ModelState

5.1 Data Validation

In an ASP.NET MVC application, users interact with the application in the following ways:

- Typing a URL on the browser.
- Clicking a link on the application.
- Submitting a form provided by the application as a view.

In all the preceding interactions, you as a developer need to ensure that any data sent by the user to the application is valid. You can validate user data by including validation logic in your application. The validation logic should first analyze whether or not the user specified data is valid and as expected by the application. Next, the validation logic should accordingly provide feedback to the user so that the user can identify and rectify any invalid input, before they resubmit the data.

5.1.1 Data Validation Workflow

Consider the scenario of an online shopping store that allows users to browse and buy products online. However, before buying a product, the user needs to register with the online store. During the registration process, the user needs to enter details, such as name, password, gender, age, and address.

The registration details, such as name, password, gender, age, and address, should be in the correct format and not left empty. Moreover, the age entered by the customer in the registration form should be equal to or more than 18 years. In such a situation, when the user submits the form, the details of the user must be validated. If the validation criteria are valid, the registration data is accepted. Otherwise, an error message is displayed to the customer. Such type of validation can be implemented on a Website by using data validation.

Before you start implementing data validation, it is important to understand how validation works in the MVC Framework. The MVC Framework implements a validation workflow to validate user data. The validation workflow starts when the user specified data arrives in the server. In the server, the request validation process of the MVC Framework performs validation of the data. This validation process first checks whether the request is some form of attack, such as Cross Site Scripting (CSS). If the process identifies a request as an attack, it throws an exception. Else, the process checks the request data against the validation requirements of the application. If all data meets the validation requirements, the process forwards the request to the application for further processing. If one or more data does not meet the validation requirements, the process sends back a validation error message as a response.

5.1.2 Manual Validation

In an ASP.NET MVC application, you can manually validate data in the controller action that accepts user data for some kind of processing. To manually validate data, you can implement simple validation routines, such as a routine to check that the length of a password submitted through a registration form exceeds more than seven characters.

Code Snippet 1 shows a manual validation implemented in an action method.

Code Snippet 1:

```
public class HomeController : Controller
{
    PublicActionResultIndex()
    {
        return View();
    }

    [HttpPost]
    public ActionResultIndex(User model)
    {
        string modelPassword = model.password;
        if (modelPassword.Length < 7)
        {
            return View();
        }
        else
        {
            /*Implement registration process*/
            return Content("You have successfully registered");
        }
    }
}
```

This code creates a `HomeController` controller class with two `Index()` action methods. The first `Index()` method returns the corresponding view. When the user submits a form displayed by the view, the `HttpPost` version of the action method receives a `User` model object that represents the user submitted data. This action method validates whether or not, the length of the password property is greater than seven. If the password length is less than seven, the action method returns back the view, else the action method returns a success message.

You can also use regular expression while performing manual validations. For example, you can use a regular expression to check whether the user submitted e-mail id is in the correct format, such as `abc@abc.com`.

Code Snippet 2 shows a manual validation implemented in an action method using regular expression.

Code Snippet 2:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Index(User model)
    {
        string modelEmailId = model.email;

        string regexPattern = @"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}";

        if (System.Text.RegularExpressions.Regex.IsMatch(modelEmailId,
            regexPattern))
        {
            /*Implement registration process*/
            return Content("You have successfully registered");
        }
        else
        {
            return View();
        }
    }
}
```

This code creates a `HomeController` controller class with two `Index()` action methods. The first `Index()` method returns the corresponding view. When the user submits a form displayed by the view, the `HttpPost` version of the action method receives a `User` model object that represents the user submitted data. This action method uses the `regexPattern` variable to store a regular expression that specifies a valid e-mail id format. The action method then, uses the `.IsMatch()` method of the `System.Text.RegularExpressions.Regex` class to validate whether or not, the `email` property is in valid format. If the value of the `email` property is valid, the action method returns back the view, else the action method returns a success message.

5.2 Data Annotation

The MVC Framework provides several data annotations that you can apply as attributes to a model. These data annotations implement tasks that are commonly required across applications. For example, instead of writing code to validate the length of a password in an action method, you can use the `StringLength` annotation in the model to perform the validation. When you use an annotation in a model, the request handling process of the MVC framework executes the logic of the annotation before passing the model object to the controller action method.

Some of the important annotations that you can use in the models of an ASP.NET MVC application are as follows:

- Required
- StringLength
- RegularExpression
- Range
- Compare
- DisplayName
- ReadOnly
- DataType
- ScaffoldColumn

5.2.1 Required Annotation

The `Required` data annotation specifies that the property, with which this annotation is associated. This means that the value of the property cannot be left blank. This attribute raises a validation error if the property value is null or empty. Following is the syntax of the `Required` data annotation:

Syntax:

```
[Required]
public string <property_name>;
where,
property_name: Is the name of a model property.
```

Code Snippet 3 shows use of the `Required` annotation in the properties of a `User` model.

Code Snippet 3:

```
public class User
{
    public long Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string Password { get; set; }

    [Required]
    public string ReenterPassword { get; set; }

    [Required]
    public int Age { get; set; }

    [Required]
    public string Email { get; set; }
}
```

In this code, the `Required` attribute is specified for the `Name`, `Password`, `ReenterPassword`, `Age`, and `Email` properties of the `User` model.

Once you use the `Required` attributes in the model properties, you need to first use the `Html.ValidationSummary()` helper method passing `true` as the parameter. This method is used to display validation messages on the page. Then, for each field, you need to use the `Html.ValidationMessageFor()` helper method passing the lambda expression to associate the method with a model property. This method returns the validation error message for the associated property as HTML.

Code Snippet 4 shows the view that uses helper methods to display validation messages.

Code Snippet 4:

```
@model MVCModelDemo.Models.User
@{
    ViewBag.Title = "User Form Validation";
}
<h2>User Form</h2>
@using (Html.BeginForm()) {
```

```
@Html.ValidationSummary(true)



@Html.LabelFor(model => model.Name)



@Html.EditorFor(model => model.Name)
@Html.ValidationMessageFor(model => model.Name)



@Html.LabelFor(model => model.Password)



@Html.EditorFor(model => model.Password)
@Html.ValidationMessageFor(model => model.Password)



@Html.LabelFor(model => model.ReenterPassword)



@Html.EditorFor(model => model.ReenterPassword)
@Html.ValidationMessageFor(model => model.ReenterPassword)



@Html.LabelFor(model => model.Age)



@Html.EditorFor(model => model.Age)
@Html.ValidationMessageFor(model => model.Age)



@Html.LabelFor(model => model.Email)


```

```

<div>
    @Html.EditorFor(model => model.Email)
    @Html.ValidationMessageFor(model => model.Email)
</div>
<p>
    <input type="submit" value="Submit" />
</p>
}

```

In this code, the `Html.ValidationSummary(true)` helper method displays validation messages as a list. Then, for each UI field, the `Html.ValidationMessageFor()` helper method is used to validate the corresponding property of the model.

When you use the `Required` annotation on a model property and the user submits a form without specifying a value for that property, the MVC Framework displays default validation error messages for the fields that fails validation.

Figure 5.1 shows the default validation error messages, when the user clicks the **Submit** button without specifying any values in the fields.

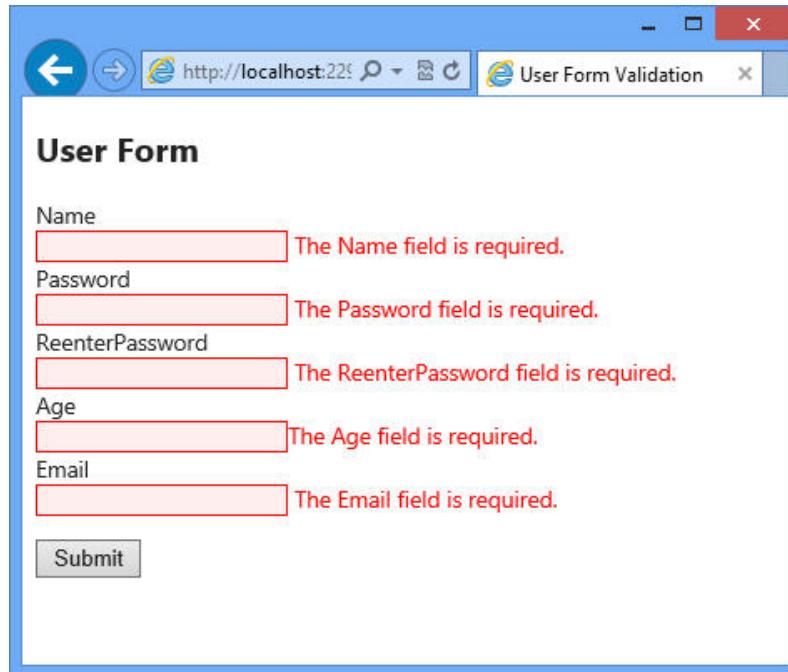


Figure 5.1: Default Validation Error Messages

You can also specify a custom validation message for the `Required` annotation. The syntax to specify a custom validation message for the `Required` annotation is as follows:

Syntax:

```
[Required(ErrorMessage = <error-message>)]
```

where,

`error-message`=: Is the custom error message that you want to display.

Code Snippet 5 shows a `User` model with the `Required` annotations with custom validation messages applied to its properties.

Code Snippet 5:

```
public class User
{
    public long Id { get; set; }

    [Required(ErrorMessage = "Please enter your name.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Please enter your password.")]
    public string Password { get; set; }

    [Required(ErrorMessage = "Please re-enter your password.")]
    public string ReenterPassword { get; set; }

    [Required(ErrorMessage = "Please enter your age.")]
    public int Age { get; set; }

    [Required(ErrorMessage = "Please enter your Email-ID.")]
    public string Email { get; set; }
}
```

Figure 5.2 shows the custom validation messages when the user tries to submit the form without specifying any values.

The screenshot shows a web browser window titled "User Form Validation". Inside, there's a form titled "User Form" with five input fields:

- Name:** An input field with a red border and the placeholder "Please enter your name."
- Password:** An input field with a red border and the placeholder "Please enter your password."
- ReenterPassword:** An input field with a red border and the placeholder "Please re-enter your password."
- Age:** An input field with a red border and the placeholder "Please enter your age."
- Email:** An input field with a red border and the placeholder "Please enter your Email-ID."

Below the inputs is a "Submit" button.

Figure 5.2: Custom Validation Error Messages

5.2.2 StringLength Annotation

You can use the `StringLength` annotation to specify the minimum and maximum lengths of a string field. For example, you use the `StringLength` annotation in a password field to specify that the length of a password must be between five to nine characters. If the user submits a password outside the specified range, the `StringLength` annotation raises a validation error.

Following is the syntax for `StringLength` annotation:

Syntax:

```
[StringLength(<max_length>, MinimumLength= <min_length>)]
```

where,

`max_length`: Is an integer value that specifies the maximum allowed length.

`min_length`: Is an integer value that specifies the minimum allowed length.

Code Snippet 6 shows a User model with the `StringLength` annotations applied to its `Password` and `ReenterPassword` properties.

Code Snippet 6:

```
public class User
{
    public long Id { get; set; }

    [Required(ErrorMessage = "Please enter your name.")]
    public string Name { get; set; }

    [StringLength(9, MinimumLength = 4)]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [StringLength(9, MinimumLength = 4)]
    [DataType(DataType.Password)]
    public string ReenterPassword { get; set; }

    [Required(ErrorMessage = "Please enter your age.")]
    public int Age { get; set; }

    [Required(ErrorMessage = "Please enter your Email-ID.")]
    public string Email { get; set; }
}
```

In this code, the `StringLength` annotation specifies that the maximum length of the `Password` and `ReenterPassword` properties is set to 9 and the minimum length is set to 4.

Whenever the user specified values for these fields are out of this specified range, a validation error message is displayed.

Figure 5.3 shows the validation messages when the user specified value in the Password and ReenterPassword fields is out of the specified range.

The screenshot shows a web browser window with the URL <http://localhost:2290/>. The title bar says "User Form Validation". The main content area is titled "User Form". It contains several input fields: "Name" (value: "mark"), "Password" (value: "", highlighted in red), "ReenterPassword" (value: "", highlighted in red), "Age" (value: "25"), and "Email" (value: "mark@mvcexample.com"). Below the inputs is a "Submit" button. Red validation messages are displayed next to the password and re-entered password fields: "The field Password must be a string with a minimum length of 4 and a maximum length of 9." and "The field ReenterPassword must be a string with a minimum length of 4 and a maximum length of 9."

Figure 5.3: Validation Error Messages

5.2.3 RegularExpression Annotation

You can use the `RegularExpression` annotation to accept user input in a specific format. For example, when the user specifies an e-mail address, it should contain @ character in between the user name and the domain name. To accept and validate such values, you can use the `RegularExpression` annotation. This annotation allows you to match a text string with a search pattern that contains one or more character literals, operators, or constructs. Following is the syntax for `RegularExpression` annotation:

Syntax:

`[RegularExpression(<pattern>)]`

where,

`<pattern>`: Is the specified format according to which you want user input.

Code Snippet 7 shows a `User` model with the `RegularExpression` annotations applied to its `Email` property.

Code Snippet 7:

```
public class User
{
    public long Id { get; set; }
```

```
[Required(ErrorMessage = "Please enter your name." )]

public string Name { get; set; }

    [StringLength(9, MinimumLength = 4) ]

public string Password { get; set; }

    [StringLength(9, MinimumLength = 4) ]

public string ReenterPassword { get; set; }

    [Required (ErrorMessage = "Please enter your age." )]

public int Age { get; set; }

    [RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}", ErrorMessage = "Email is not valid." )]

public string Email { get; set; }

}
```

In this code, the regular expression `A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}` defines the format of an e-mail address. It is divided in three parts where:

Part 1: `[A-Za-z0-9._%+-]+`

Part 2: `[A-Za-z0-9.-]+`

Part 3: `[A-Za-z]{2,4}`

The first part of the regular expression specifies the characters and it ranges within square brackets that can appear. Then, the + sign between the first and second part indicates that these parts can consist of one or more characters of the types specified within the square brackets preceding the + sign. The third part contains `{2,4}` at the end that indicates that this part can include 2-4 characters.

Figure 5.4 shows the validation message that is displayed when the user specified value in the Email field is not in a valid format as specified using regular expression.

The screenshot shows a web browser window titled "User Form Validation" with the URL "http://localhost:225". The form contains five input fields: Name (Mark), Password, ReenterPassword, Age (16), and Email (mark@com). The Email field is highlighted with a red border, and the text "Email is not valid." is displayed next to it. Below the form is a "Submit" button.

Figure 5.4: Validating the Email Field

5.2.4 Range Annotation

You can use the `Range` annotation to specify the minimum and maximum constraints for a numeric value. Consider a scenario, where you need user data for the `Age` field. Now, you want that the user specified age should be in between 18 to 35. In such scenario, you can use the `Range` annotation to accept user input.

Syntax:

```
[Range (<minimum_range>, <maximum_range>) ]
```

where,

`minimum_range`: Is a numeric value that specifies the minimum value for the range.

`maximum_range`: Is a numeric value that specifies the maximum value for the range.

Code Snippet 8 shows a `User` model with the `Range` annotations applied to its `Age` property.

Code Snippet 8:

```
public class User
{
    public long Id { get; set; }
```

```
[Required(ErrorMessage = "Please enter your name.")]  
public string Name { get; set; }  
  
[StringLength(9, MinimumLength = 4)]  
public string Password { get; set; }  
  
[StringLength(9, MinimumLength = 4)]  
public string ReenterPassword { get; set; }  
  
[Range(18, 60, ErrorMessage = "The age should be between 18 and 60.")]  
public int Age { get; set; }  
  
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",  
ErrorMessage = "Email is not valid.")]  
public string Email { get; set; }  
}
```

This code shows use of the `Range` annotations to the `Age` property.

Figure 5.5 shows the validation error message that is displayed when the user specified age is not in the specified range.

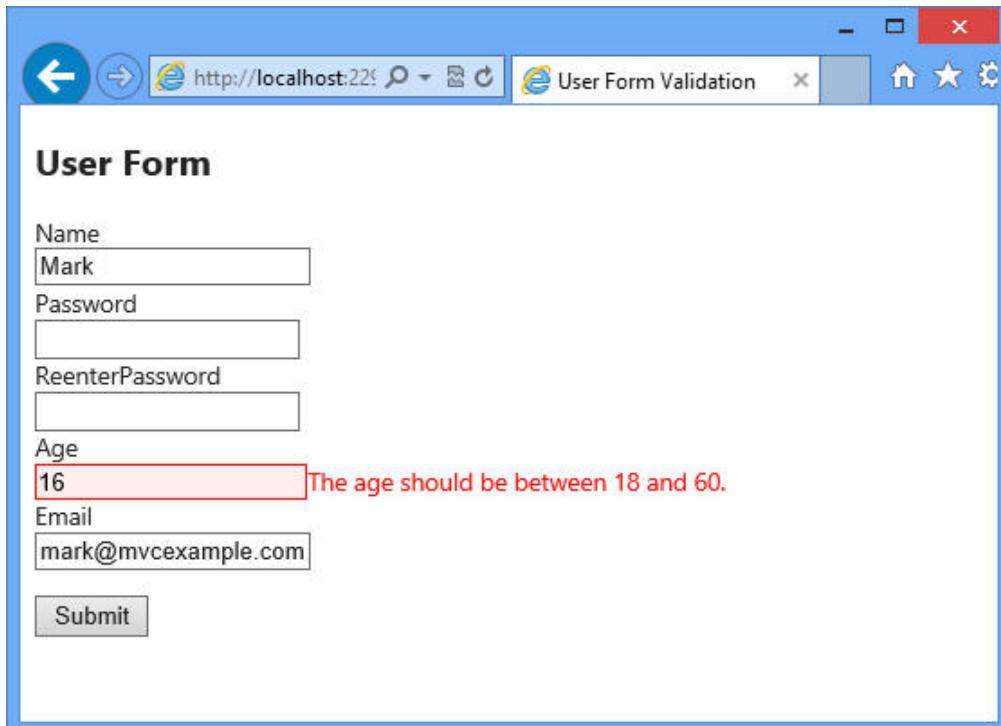


Figure 5.5: Validating the Age Field

5.2.5 Compare Annotation

You can use the `Compare` annotation to compare values in two fields. Consider a scenario, where you want to accept the same value in two different fields, such as `Password` and `ReenterPassword` in a form. For that, you need to match the user specified values in each of the fields in the form. In such scenario, you can use the `Compare` annotation that ensures that the two properties on a model object have the same value.

Code Snippet 9 shows a `User` model with the `Compare` annotation applied to its `ReenterPassword` property.

Code Snippet 9:

```
public class User
{
    public long Id { get; set; }

    [Required(ErrorMessage = "Please enter your name.")]
    public string Name { get; set; }

    [StringLength(9, MinimumLength = 4)]
    public string Password { get; set; }

    [StringLength(9, MinimumLength = 4)]
    [Compare("Password", ErrorMessage = "The specified passwords do not match with
the Password field.")]
    public string ReenterPassword { get; set; }

    [Range(18, 60, ErrorMessage = "The age should be between 18 and 60.")]
    public int Age { get; set; }

    [RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
ErrorMessage = "Email is not valid.")]
    public string Email { get; set; }
}
```

This code uses the `Compare` annotation to check that the `ReenterPassword` field contains the same values as that of the `Password` field.

Figure 5.6 shows the validation error message that is displayed when the user do not enter the same value in both the Password and ReenterPassword fields.

The screenshot shows a web browser window titled "User Form Validation". Inside the window, there is a form titled "User Form". The form contains the following fields:

- Name: A text input field containing "Mark".
- Password: An empty text input field.
- ReenterPassword: An empty text input field, which is highlighted with a red border and contains the validation error message "The specified passwords do not match with the Password field.".
- Age: A text input field containing "25".
- Email: A text input field containing "mark@mvceexample.com".

Below the form is a "Submit" button.

Figure 5.6: Validating the ReenterPassword Field

5.2.6 DisplayName Annotation

When you use the `@Html.LabelFor()` helper method in a strongly typed view, the method displays a label with a text whose value is the corresponding property name. For example, for the `ReenterPassword` property, the `Html.LabelFor()` helper method will display a `ReenterPassword` label. However, you can explicitly state the text that the `@Html.LabelFor()` method should display using the `DisplayName` annotation on the model property.

Following is the syntax for `DisplayName` annotation:

Syntax:

```
[DisplayName(<text>)]
```

where,

`text`: Is the text to be displayed for the property.

Code Snippet 10 shows a User model with the DisplayName annotation applied to its Name, ReenterPassword, and Email properties.

Code Snippet 10:

```
Public class User
{
    public long Id { get; set; }

    [DisplayName("User Name")]
    public string Name { get; set; }

    public string Password { get; set; }

    [DisplayName("Re-enter Password")]
    public string ReenterPassword { get; set; }

    public int Age { get; set; }

    [DisplayName("Email- ID")]
    public string Email { get; set; }
}
```

This code applies the DisplayName annotation to the Name, ReenterPassword, and Email properties.

Figure 5.7 shows the user friendly display name for the Name, ReenterPassword, and Email model properties.

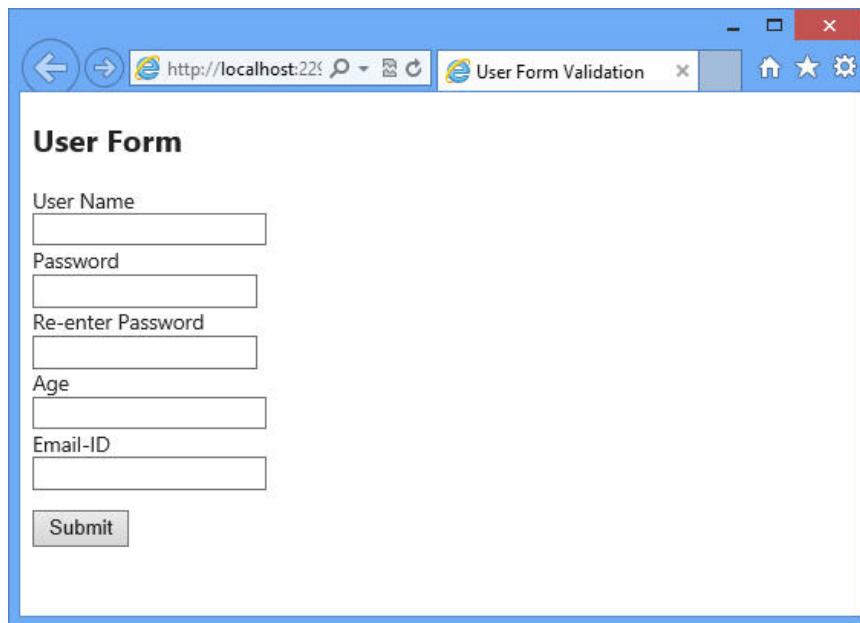


Figure 5.7: Output of Using the DisplayName Annotation

5.2.7 `ReadOnly` Annotation

You can use the `ReadOnly` annotation to display read-only fields on a form. Consider an example of a form that allows user to specify the product details to purchase from an online store. The model for this form will contain a `DiscountedAmount` property that will store the price after a discount is provided by the application logic. Therefore, you do not require the `DiscountedAmount` property to be displayed as an editable field in the form. You can achieve this by using the `ReadOnly` annotation. By using this annotation, you can instruct the default model binder not to set the `DiscountedAmount` property with a new value from the request.

Following is the syntax for `ReadOnly` annotation:

Syntax:

```
[ReadOnly(<boolean_value>)]
```

where,

`boolean_value`: Is a boolean value which can be either `true` or `false`. A `true` value indicates that the default model binder should not set a new value for the property. A `false` value indicates that the default model binder should set a new value for the property based on the request.

Code Snippet 11 shows how to use the `ReadOnly` annotation.

Code Snippet 11:

```
[ReadOnly(true)]
public int DiscountedAmount { get; set; }
```

This code uses the `ReadOnly` annotation passing `true` as the value for the `DiscountedAmount` property.

5.2.8 `DataType` Annotation

You can use the `DataType` annotation to provide information about the specific purpose of a property at runtime. For example, you can display some special character instead of plain text, when the user enters their password in the `Password` field.

Following is the syntax for `DataType` annotation:

Syntax:

```
[DataType(DataType.<value>)]
```

where,

`value`: Is a member of `DataType` enumeration.

Similarly, you can specify that the `Html.EditorFor()` should display a multi-line text area instead of a single-line text field for an `Address` property using the `DataType` annotation.

Code Snippet 12 shows applying the `DataType` annotation to the `Password` and `Address` properties of a model.

Code Snippet 12:

```
[DataType(DataType.Password)]
public string Password { get; set; }

[DataType(DataType.MultilineText)]
public string Address { get; set; }
```

In this code, the `DataType` annotation is first applied to the `Password` property. This instructs the view to display a password field masks the user entered password. Next, the `DataType` annotation is applied to the `Address` property. This instructs the view to display a multi-line text area for the `Address` property.

Figure 5.8 shows the view that displays the fields for the `Password` and `Address` properties.

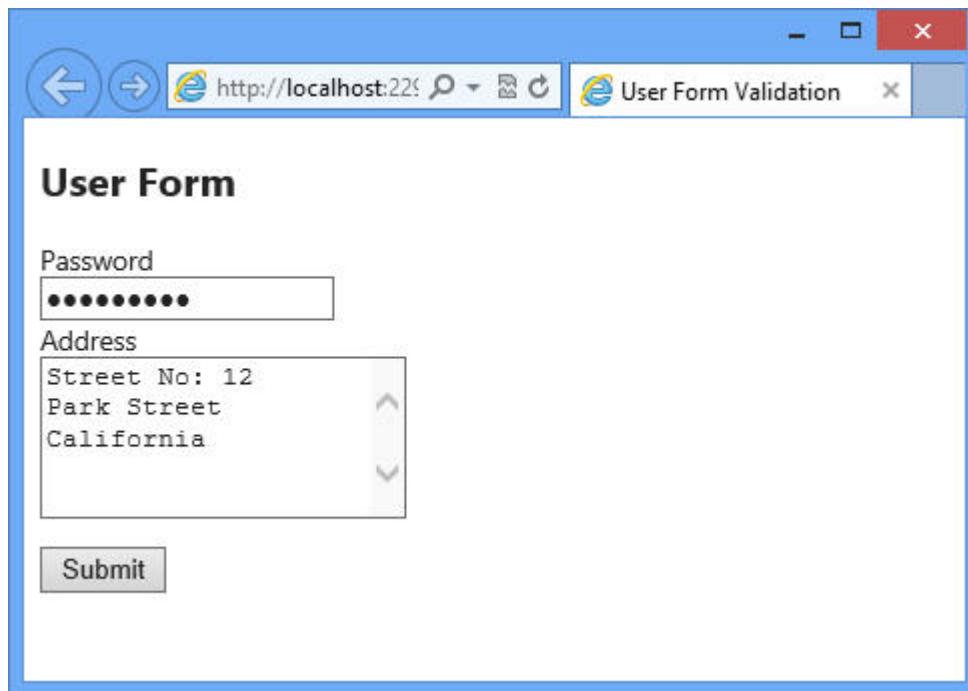


Figure 5.8: Displaying the Password and Address Fields

5.2.9 ScaffoldColumn Annotation

You have already learned how to use the scaffolding feature of Visual Studio .NET 2013 to automatically generate strongly typed views based on a model. You have used the Create scaffolding template to create a form for the `User` model.

When you use scaffolding using the Create template, the view by default, will create UI fields for all the properties of the model. However, you might need to ensure that the template does not create UI fields for certain properties. For example, in a `User` model, you will not want the Create scaffolding template to create a field for the `Id` property as the value of this property will be generated by the application instead of accepting from user. In such scenarios, you can use the `ScaffoldColumn` annotation passing a `false` value.

Code Snippet 13 shows use of the `ScaffoldColumn` annotation.

Code Snippet 13:

```
[ScaffoldColumn (false)]
public long Id { get; set; }
```

In this code, the `ScaffoldColumn` annotation with a `false` value instructs the Create scaffolding template not to create a UI field in the view for the `Id` property.

5.3 ModelState Validation

`ModelState` is a class in the `System.Web.Mvc` namespace that encapsulate the state of model binding in an application. A `ModelState` object stores the values that the user submits to update a model and any validation errors. You can check whether the state of a model is valid by using the `ModelState.IsValid` property. For example, consider that the user submits an e-mail id in an incorrect format. Subsequently, the model binding process of the user entered e-mail id with the `email` property of the model class fails and a validation error results.

In the action, you can use the `ModelState.IsValid` property to check the state for this model, which in this example will be `false`. You can therefore, return the view to display a validation error message by using the `Html.ValidationMessage()` helper method. If the model binding succeeds, the `ModelState.IsValid` property returns `true` and you can accordingly perform the required function with the model data.

Code Snippet 14 shows how to use the `ModelState.IsValid` property.

Code Snippet 14:

```
public ActionResult Index (User model)
{
    if (ModelState.IsValid)
    {
        /*Perform required function with the model data*/
        return Content ("You have successfully registered");
    }
}
```

```
    }  
    else  
    return View();  
}
```

In this code, the `ModelState.IsValid` is used to check the state of the `User` model. If the property returns true, a success message is displayed. Else, the view is return so that the user can enter valid values.

5.4 Check Your Progress

1. Which of the following statements about the data validation workflow are true?

(A)	The data validation workflow is part of the MVC Framework to validate user input data.
(B)	The data validation workflow starts when the user input data arrives on the server.
(C)	The data validation workflow starts when the user enters data in a form.
(D)	The data validation workflow implements client-side validation process.
(E)	The data validation workflow is part of the MVC Framework to validate data that the application sends as response.

(A)	A, B	(C)	A, D
(B)	C, D	(D)	E, D

2. Match the descriptions against their corresponding data annotation.

Data Annotations		Description	
(A)	StringLength	(1)	Specifies that the property, with which this annotation is associated, is a required property.
(B)	RegularExpression	(2)	Specifies the minimum and maximum lengths of a string field.
(C)	Required	(3)	Accepts user input in a specific format.
(D)	Range	(4)	Compare values in two different fields.
(E)	Compare	(5)	Specifies the minimum and maximum constraints for a numeric value.

(A)	A-3, B-4, C-1, D-2, E-5	(C)	A-3, B-4, C-5, D-1, E-2
(B)	A-2, B-3, C-1, D-5, E-4	(D)	A-5, B-4, C-1, D-3, E-2

3. Which of the following method validates whether the value of a specified property matches a regular expression?

(A)	IsMatch()
(B)	Html.ValidationSummary()
(C)	Html.ValidationMessageFor()
(D)	@Html.LabelFor()
(E)	Html.ValidationMessage()

(A)	B	(C)	D
(B)	C	(D)	A

4. Which of the following annotation will you use in a model property to notify the view to display a text area for that property?

(A)	ReadOnly
(B)	Range
(C)	DisplayName
(D)	DataType
(E)	ScaffoldColumn

(A)	B	(C)	D
(B)	C	(D)	A

5. Which of the following code correctly examines the state of a model in a controller's action?

(A)	<pre>public ActionResult Index(User model) { if (ModelState.IsValid) { /*Perform required function with the model data*/ return Content("Welcome user"); } }</pre>
-----	--

(A)	<pre> else return View(); } </pre>
(B)	<pre> public ActionResult Index(User model) { if (!ModelState.IsValid) { /*Perform required function with the model data*/ return Content("Welcome user"); } else return View(); } </pre>
(C)	<pre> public ActionResult Index(User model) { if (ModelState.IsValid()) { /*Perform required function with the model data*/ return Content("Welcome user"); } else return View(); } </pre>
(D)	<pre> public ActionResult Index(User model) { if (!ModelState.IsValid) { return View(); } } </pre>

<pre>(D) } else { /*Perform required function with the model data*/ return Content("Welcome user"); } }</pre>
--

<pre>(E) public ActionResult Index(User model) { if (model.IsValid) { /*Perform required function with the model data*/ return Content("Welcome user"); } else return View(); }</pre>
--

5.4.1 Answers

(1)	A
(2)	B
(3)	D
(4)	C
(5)	C



Summary

- The MVC Framework implements a validation workflow to validate user data.
- In an ASP.NET MVC application, you can manually validate data in the controller action.
- The MVC Framework provides several data annotations that you can apply as attributes to the properties of a model.
- The Required annotation when applied to a property validates that a value is specified for that property.
- The RegularExpression annotation when applied to a property validates that a value specified for that property, matches the specified regular expression.
- The @Html.LabelFor() helper method when used in a strongly typed view displays a label with a text whose value is the corresponding property name.
- ModelState is a class in the System.Web.Mvc namespace that encapsulate the state of model binding in an application.

Session - 6

Data Access

Welcome to the Session, **Data Access**.

In an ASP.NET MVC application, you can use an Object Relationship Mapping (ORM) framework to simplify the process of accessing data from the application. You can use the Entity Framework, which is an ORM framework in your applications. Entity Framework provides the database-first and code-first approaches to manage data related to an application.

The System.Data.Entity namespace of the Entity Framework provides classes that you can use to synchronize between the model classes and its associated database. This namespace also provides the DbContext class that coordinates with Entity Framework and allows you to query and save application data in the database.

In an ASP.NET MVC application, you can use LINQ to create data-source-independent queries to implement data access in an application. You can write LINQ queries in any .NET Framework supported programming language.

In this Session, you will learn to:

- ➔ Define and describe the Entity Framework
- ➔ Explain how to work with the Entity Framework
- ➔ Define and describe how to initialize a database with sample data
- ➔ Explain how to use LINQ queries to perform database operations

6.1 Entity Framework

Most Web applications need to persist and retrieve data that might be stored in some data source, such as a relational database, XML file, or spreadsheet. In such application, data is usually represented in the form of classes and objects. However, in a database, data is stored in the form of tables and views. Therefore, an application in order to persist and retrieve data first needs to connect with the data store. The application must also ensure that the definitions and relationships of classes or objects are mapped with the database tables and table relationships. Finally, the application needs to provide the data access code to persist and retrieve data.

To address data access requirements of ASP.NET MVC application, you can use an ORM framework. An ORM framework simplifies the process of accessing data from applications. An ORM framework performs the necessary conversions between incompatible type systems in relational databases and object-oriented programming languages. The Entity Framework is an ORM framework that ASP.NET MVC applications can use.

The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application. EDM allows you to handle data access logic by programming against entities without having to worry about the structure of the underlying data store and how to connect with it.

For example, in an order placing operation of a customer relationship management application, a programmer using the EDM can work with the Customer and Order entities in an object-oriented manner without writing database connectivity code or SQL-based data access code.

Figure 6.1 shows the role of EDM in the Entity Framework architecture.

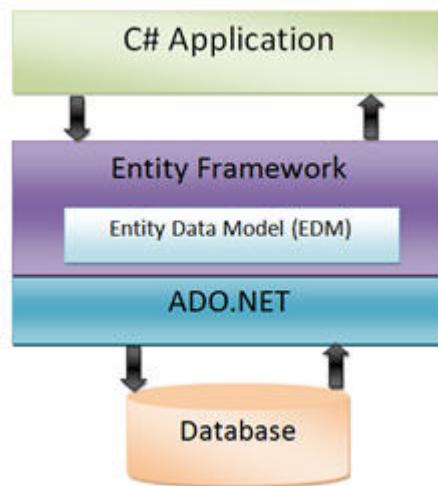


Figure 6.1: Entity Framework Architecture

Entity Framework eliminates the need to write most of the data-access code that otherwise needs to be written. It uses different approaches, such as database-first and code-first to manage data related to an application.

6.1.1 Database-first Approach

In this approach, the Entity Framework creates model classes and properties corresponding to the existing database objects, such as tables and columns. This approach is applicable in scenarios where a database already exists for the application.

Figure 6.2 shows the database-first approach.

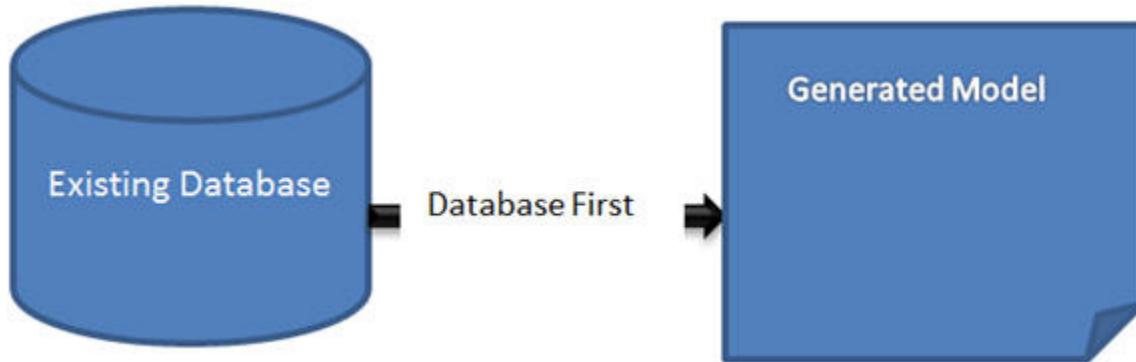


Figure 6.2: Database-first Approach

6.1.2 Code-first Approach

In this approach, the Entity Framework creates database objects based on model classes that you create to represent application data. This is the most common approach implemented in ASP.NET MVC Framework.

This approach allows you to develop your application by coding model classes and properties and delegate the process of creating the database objects to the Entity Framework. These classes and properties will later correspond to tables and columns in a database.

Figure 6.3 shows the code-first approach.

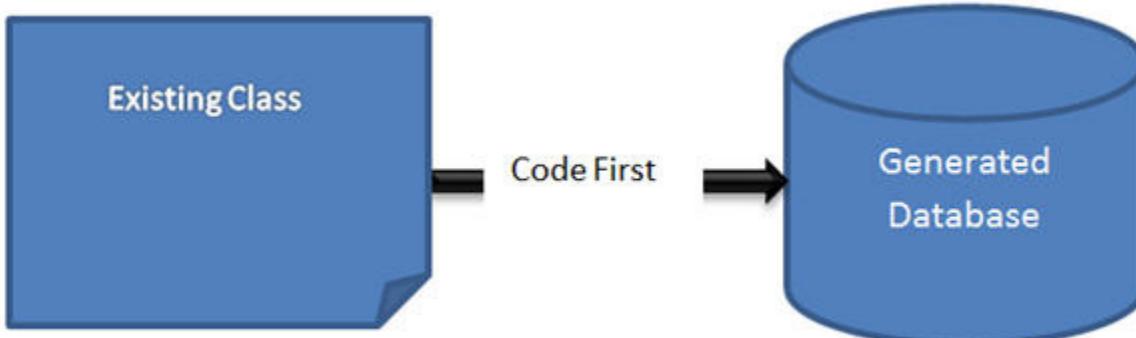


Figure 6.3: Code-first Approach

6.2 Working with Entity Framework

The Entity Framework uses different approaches to manage data related to objects. The code-first approach allows you to define your own model by creating custom classes. Then, you can create database based on the models. There are certain conventions that you should follow for the code-first approach.

6.2.1 Implementing Code-first Approach

The code-first approach allows you to provide the description of a model by using the C# classes. Based on the class definitions, the code-first conventions detect the basic structure for the model. The `System.Data.Entity.ModelConfiguration.Conventions` namespace provides several code-first conventions that enables automatic configuration of a model. Some of these conventions are as follows:

- **Table naming convention:** When you have created an object of the `User` model and need to store its data in the database, Entity Framework by default creates a table named `Users`.
- **Primary key convention:** When you create a property named `UserId` in the `User` model, the property is accepted as a primary key. In addition, the Entity Framework sets up an auto-incrementing key column to hold the property value.

Consider a scenario that you have created a `Customer` model that contains the `Id` and `Name` properties.

Code Snippet 1 shows the `Customer` model.

Code Snippet 1:

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

In this code, you have created two properties named `Id` and `Name` in the model class named `Customer`. Here, the `Id` property will be inferred by the Entity Framework as a primary key.

- **Relationship convention:** Entity Framework provides different conventions to identify a relationship between two models. You can use navigational properties in order to define relationship between two models. You should define a foreign key property on types that represents dependent objects.

Consider a scenario where, you are developing an online shopping store. For the application, you have created two model classes named `Customer` and `Order`. Now, you need to declare properties in each class that allows navigating to the properties of another class. You can then, define the relationship between these two classes.

Code Snippet 2 creates the `Customer` model class.

Code Snippet 2:

```
public class Customer
{
    public int CustId { get; set; }
    public string Name { get; set; }
    // Navigation property
    public virtual ICollection<Order> Orders { get; set; }
}
```

This code creates a model named `Customer` that contains two properties named, `CustId` and `Name`.

Code Snippet 3 creates the `Order` model class.

Code Snippet 3:

```
public class Order
{
    public int Id { get; set; }
    public string ProductName { get; set; }
    public int Price { get; set; }
    // Foreign key
    public int CustId { get; set; }
    // Navigation properties
    public virtual Customer cust { get; set; }
}
```

In this code, the `Orders` is the navigational property in the `Customer` class and `cust` is the navigational property in the `Order` class. These two properties are known as navigational properties because they allow us to navigate to the properties of another class.

For example, you can use the `cust` property to navigate to the orders associated with that customer. In the `Customer` class the `Orders` navigational property is declared as a collection, as one customer can place multiple orders. This indicates a one-to-many relationship between the `Customer` and `Order` classes. The `CustId` property in the `Order` class is inferred as the foreign key by Entity Framework.

6.2.2 The `DbContext` Class

The `System.Data.Entity` namespace of the ASP.NET MVC Framework provides a `DbContext` class. After creating a model class, you can use the `DbContext` class to define the database context class. This

class coordinates with Entity Framework and allows you to query and save the data in the database.

The database context class uses the `DbSet <T>` type to define one or more properties. In the type, `DbSet <T>`, `T` represents the type of an object that needs to be stored in the database.

Code Snippet 4 shows how to use the `DbContext` class.

Code Snippet 4:

```
public class OLShopDataContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

In this code, a database context class named `OLShopDataContext` is created that derives from the `DbContext` class. This class creates the `DBSet` property for both, the `Customer` class and the `Product` class.

6.3 Initializing a Database with Test Data

While developing an ASP.NET MVC Web application, you might need to change the model classes to implement various new features. This also requires maintaining the database related to the model based on the changes made in the model class. So while modifying the model classes, you should ensure that any changes in the model are reflected back in the database. To maintain the synchronization between the model classes and its associated database, you need to recreate databases.

Entity Framework provides the `System.Data.Entity` namespace that contains the following two classes to recreate databases:

- ➔ **`DropCreateDatabaseAlways`:** Allows recreating an existing database whenever the application starts.
- ➔ **`DropCreateDatabaseIfModelChanges`:** Allows recreating an existing database whenever the associated model class changes.

Based on your requirements, you can use one of these two classes in your application to recreate a database.

You can use one of these two classes while calling the `SetInitializer()` method of the `System.Data.Entity.Database` namespace. For example, consider that you want to recreate the database for an application whenever the application starts. To recreate the database whenever the application starts, you can use the `DropCreateDatabaseAlways` class while calling the `SetInitializer()` method inside the `Global.asax.cs` file.

Code snippet 5 shows using the `DropCreateDatabaseAlways` class inside the `Application_Start()` method of the `Global.asax.cs` file.

Code Snippet 5:

```
Database.SetInitializer(new
DropCreateDatabaseAlways<ShopDataContext>());
```

In this code, the `DropCreateDatabaseAlways` class is used while calling the `SetInitializer()` method to ensure that the existing database is recreated whenever the application starts.

On the other hand, you can use the `DropCreateDatabaseIfModelChanges` class to recreate a database only when the model changes.

For example, you have made some changes in one of the model class of the online shopping store application.

Code Snippet 6 shows using the `DropCreateDatabaseIfModelChanges` class inside the `Application_Start()` method of the `Global.asax.cs` file.

Code Snippet 6:

```
Database.SetInitializer(new DropCreateDatabaseIfModelChanges<ShopDataContext>());
```

In this code, the `DropCreateDatabaseIfModelChanges` class is used while calling the `SetInitializer()` method to ensure that the existing database is recreated whenever the model changes.

You can also instruct the MVC Framework to populate a database with sample data for an application. This can be achieved by creating a class that derives from either the `DropCreateDatabaseIfModelChanges` class or the `DropCreateDatabaseAlways` class. In this class, you need to override the `Seed()` method that enables you to define the initial data for the application.

Code Snippet 7 shows the `MyDbInitializer` model class that uses the `Seed()` method to insert some sample data in the `Customers` database.

Code Snippet 7:

```
public class MyDbInitializer : DropCreateDatabaseIfModelChanges<OLShopDataContext>
{
    protected override void Seed(OLShopDataContext context)
    {
        context.Customers.Add(new Customer() { Name = "John Parker", Address = "Park Street", Email = "john@mvcexample.com" });
        base.Seed(context);
    }
}
```

In the code, the `MyDbInitializer` class is derived from the `DropCreateDatabaseIfModelChanges` class. Then, the `Seed()` method is overridden to define the initial data for the `Customer` model.

Once you have defined the initial data for the customers, you need to register the `MyDbInitializer` model class in the `Global.asax.cs` file by calling the `SetInitializer()` method.

Code Snippet 8 shows the `SetInitializer()` method inside the `Application_Start()` method.

Code Snippet 8:

```
protected void Application_Start()
{
    System.Data.Entity.Database.SetInitializer(new
    MyDbInitializer());
}
```

This code uses the `SetInitializer()` method to register the `MyDbInitializer` model class.

6.4 LINQ Query

LINQ is a set of APIs that allows you to create data-source-independent queries to implement data access in an application. It has a programming model that provides the standard query syntax to query different types of data sources. In LINQ, the query is written in any .NET Framework supported programming language, such as VB or C#. The LINQ query acts on a strongly-typed collection of objects with the help of language keywords and common operators. This helps you to quickly manipulate data from the various data sources without requiring them to learn a new query language.

6.4.1 Using a Simple LINQ Query

In an ASP.NET MVC application, a LINQ query operation starts by creating a query. Once the query is created, the next operation is to execute the query. To execute a LINQ query, you need to first specify the data source. Typically, LINQ queries operate on objects that implement either the `IEnumerable<T>` or `IQueryable<T>` interface. Both these interfaces enable you to create queries to retrieve data from a specific data source.

Consider the scenario, where you need to retrieve the customer details from a `Customer` table stored in a database, named `OLSSDB`. For this, you can use an instance of the `IQueryable<T>` interface to retrieve the customer details from the data source.

Code Snippet 9 shows creating a query that retrieves customer details from the `Customer` data source.

Code Snippet 9:

```
IQueryable<Customer> q = from s in db.customers select s;
```

where,

`from:` Clause specifies the data source that contains the data.

`db`: Is an instance of the data context class provides access to the data source.

`s`: Is the range variable.

`select`: Clause specifies that each element in the result will consist of a customer object.

Once you have created a LINQ query, you need to execute it to retrieve the data and then, iterate over the result set using a `foreach` loop.

Consider the scenario of the online shopping store application, where you need to retrieve the customer details from the `Customers` table and display the details. For this, you first need to create a query then, use the `foreach` loop to execute the query and retrieve the details.

Code Snippet 10 shows using `foreach` loop to retrieve customer details.

Code Snippet 10:

```
string names = "";
IQueryable<Customer> q = from s in db.customers
    select s;
foreach (var cust in q)
{
    names = names + " " + cust.Name;
}
ViewBag.Name = names;
```

In this code, the `foreach` loop retrieves the query results and the `cust` variable stores each value one at a time. Then, the name of each customer is appended to the `string` variable, `names`. Finally, the `names` variable is assigned to the `Name` property using `ViewBag` to display the information in a view.

6.4.2 Using Advance LINQ Queries

You have already learned how to create and execute LINQ queries. In addition to retrieving data from data sources, you can also use LINQ queries to perform various operations on data stored in a data source. Some of the common operations that you perform on data using LINQ queries include the following:

- ➔ Forming Projections
- ➔ Filtering the data
- ➔ Sorting the data
- ➔ Grouping the data

In addition to simple data retrieval, programmers can use LINQ to perform various other operations, such as forming projections, filtering data, and sorting data.

→ Forming Projections

While using LINQ queries, you might only need to retrieve specific properties of a model from the data store, for example, only the Name property of the Customer model. You can achieve this by forming projections in the select clause.

Code Snippet 11 shows a LINQ query that retrieves only the customer names of the Customer model class.

Code Snippet 11:

```
public static void DisplayCustomerNames()
{
    using (Model1Container dbContext = new Model1Container())
    {
        IQueryable<String> query = from c in dbContext.Customers
                                      select c.Name;
        Console.WriteLine("Customer Names:");
        foreach (String custName in query)
        {
            Console.WriteLine(custName);
        }
    }
}
```

In the code, the `select` clause retrieves a sequence of customer names as an `IQueryable<String>` object. The `foreach` loop iterates through the result to print out the names.

Output:

Customer Names:

Alex Parker

Peter Milne

→ Filtering Data

The `where` clause in a LINQ query enables filtering data based on a Boolean condition, known as the predicate. The `where` clause applies the predicate to the `range` variable that represents the source elements and returns only those elements for which the predicate is `true`.

Code Snippet 12 uses the where clause to filter customer records.

Code Snippet 12:

```
public static void DisplayCustomerByName()
{
    using (Model1Container dbContext = new Model1Container())
    {
        IQueryable<Customer> query = from c in dbContext.Customers
where c.Name == "Alex Parker" select c;
        Console.WriteLine("Customer Information:");
        foreach (Customer cust in query)
        {
            Console.WriteLine("Customer ID: {0}, Name: {1},
Address: {2}", cust.CustomerId, cust.Name, cust.Address);
        }
    }
}
```

This code uses the `where` clause to retrieve information of the customer with the name `AlexParker`. The `foreach` statement iterates through the result to print the information of the customer.

Output:

```
Customer Information:
Customer ID: 1, Name: Alex Parker, Address: 10th Park Street, Leo Mount
```

→ Sorting the Data

You can also use LINQ queries to retrieve data and then, display it in sorted manner. To use sorting in a LINQ query, you need to use the `orderby` clause, which specifies whether the result should be displayed in either ascending or descending order.

Though, the result is displayed in the ascending order by default, you can also use the `ascending` keyword to sort in the ascending order.

Consider the scenario, where you want to display the customer name that lives in New Jersey in an ascending order. For this, you first need to retrieve the records from the `Customer` table and then, use the `ascending` keyword along with the `orderby` clause.

Code Snippet 13 shows using the `ascending` keyword to display the customer name in ascending order.

Code Snippet 13:

```
IQueryable<Customer> q = from s in dbContext.customers
                           where s.City == "New Jersey"
                           orderby s.Name ascending
                           select s;
```

This code will retrieve and then, displays the customer name who lives in New Jersey in the ascending order.

Note - You can also sort the customer name in descending order by specifying `descending` in the `orderby` clause.

→ **Grouping the Data**

You can also use LINQ query to retrieve and display the data as a group. For this, you need to use the `group` clause that allows you to group the results based on a specified key.

For example, you need to group the customers by the city name so that all the customers from a particular city are arranged in individual groups.

Code Snippet 14 shows how to group the customers according to their cities.

Code Snippet 14:

```
var q = from s in dbContext.customers
        group s by s.City;
```

This code retrieves the customer records and groups these records based on the city where the customers lives.

6.4.3 Querying Data by Using LINQ Method-based Queries

The LINQ queries used so far are created using query expression syntax. Such queries are compiled into method calls to the standard query operators, such as `select`, `where`, and `orderby`. Another way to create LINQ queries is by using method-based queries where you can directly make method calls to the standard query operator, passing lambda expressions as the parameters.

Code Snippet 15 uses the `Select` clause to project the `Name` and `Address` properties of `Customer` model class into a sequence of anonymous types.

Code Snippet 15:

```
public static void DisplayPropertiesMethodBasedQuery()
{
}
```

```

using (Model1Container dbContext = new Model1Container())
{
    var query = dbContext.Customers.Select(c => new
    {
        CustomerName = c.Name,
        CustomerAddress = c.Address
    });
    Console.WriteLine("Customer Names and Addresses:");
    foreach (var custInfo in query)
    {
        Console.WriteLine("Name: {0}, Address: {1}",
        custInfo.CustomerName, custInfo.CustomerAddress);
    }
}
}

```

This code uses the `Select` clause to project the `Name` and `Address` properties of `Customer` model class into a sequence of anonymous types.

Output:

```

Customer Names and Addresses:
Name: Alex Parker, Address: 10th Park Street, Leo Mount
Name: Peter Milne, Address: Lake View Street, Cheros Mount

```

Similarly, you can use other operators such as `Where`, `GroupBy`, `Max`, and so on through method-based queries.

6.5 LINQ Data Providers

LINQ provides a consistent programming model to create standard query expression syntax to query different types of data sources. However, different data sources accept queries in different formats. To solve this problem, there are several LINQ providers, such as LINQ to SQL, LINQ to Objects, and LINQ to XML. The LINQ queries, that you have learned till now uses the LINQ to SQL provider. This provider allows you to access SQL-complaint databases and makes data available as objects in an application.

6.5.1 LINQ to Objects

LINQ to Objects refers to the use of LINQ queries with enumerable collections, such as `List<T>` or arrays. To use LINQ to query an object collection, you need to declare a range variable. The type of the range variable should match the type of the objects in the collection.

Code Snippet 16 shows how to access data from a string array that contains the names of customers in a class.

Code Snippet 16:

```
string products="";
string[] arr=new string[] { "Laptop", "Mobile", "Jewellery" };
var query=from string product in arr
select product;
foreach (var p in query)
{
    products=products+ " " + p;
}
```

This code accesses the data from a string array that contains the names of customers.

6.5.2 LINQ to XML

You can use the LINQ to XML queries to work with XML in an application. LINQ to XML query enables you to access data that is stored in XML files.

Code Snippet 17 shows the content of an XML file named, `CustomersRecord.xml`.

Code Snippet 17:

```
<?xml version="1.0" encoding="utf-8"?>
<CustomersDetails>
    <Customer>
        <CustID>CID1001</CustID>
        <Name>Peter Jones</Name>
        <City>New York</City>
    </Customer>
    <Customer>
        <CustID>CID1002</CustID>
        <Name>Jessica Parker</Name>
    </Customer>
</CustomersDetails>
```

```
<City>London</City>
</Customer>
</CustomersDetails>
```

This code shows an XML document, named `CustomersRecord.Xml` that contains customer details.

Now, to access the data from `CustomersRecord.Xml` you can use LINQ.

Code Snippet 18 shows how to retrieve data from a XML file by using LINQ query.

Code Snippet 18:

```
string result = "";
XDocument xmlDoc = XDocument.Load("E:\\CustomerRecord.xml");
var q = from c in xmlDoc.Descendants("Customer")
        select (string)c.Element("CustID") + "-" +
               (string)c.Element("Name") + "-" + (string)c.Element("City");
foreach (string entry in q)
{
    result += entry + "|";
}
```

This code will retrieve all the customer details from the `CustomersRecord.Xml` file. The `xmlDoc.Descendants()` method returns a collection of the descendant elements for the specified element in the order they are present in the XML file.

6.6 Check Your Progress

1. Which of the following statements about Entity Framework are true?

(A)	The Entity Framework allows creating data-source-independent queries to implement data access.
(B)	The Entity Framework is an ORM framework that ASP.NET MVC applications can use.
(C)	The Entity Framework provides four approaches to manage data related to an application.
(D)	The Entity Framework is an implementation of EDM.
(E)	The Entity Framework enables you to write most of the data-access code.

(A)	A, B	(C)	A, D
(B)	C, D	(D)	B, D

2. Which of the following options in an application coordinates with Entity Framework and allows you to query and save the data in the database?

(A)	virtual keyword
(B)	DbContext class
(C)	DropCreateDatabaseAlways class
(D)	SetInitializer() method

(A)	B	(C)	D
(B)	C	(D)	A

3. Which of these statements about LINQ queries are true?

(A)	LINQ query acts on a strongly-typed collection of objects with the help of language keywords and common operators.
(B)	LINQ allows you to write query using only C# programming language.
(C)	LINQ queries operate on objects that implement only the <code>IQueryable<T></code> interface.
(D)	LINQ queries enable performing various operations on data stored in a data source.
(E)	LINQ queries instruct the MVC Framework to populate a database with sample data.

(A)	A, B	(C)	E, D
(B)	C, E	(D)	A, D

4. Which of the following options will you use to insert some sample data in a database?

(A)	add() method		
(B)	Application_Start() method		
(C)	SetInitializer() method		
(D)	Seed() method		

(A)	B	(C)	D
(B)	C	(D)	A

5. Which of the following code correctly uses the `DbContext` class?

(A)	<pre>public class MyDataContext : DbContext { public DbSet<Customer> Customers { get; set; } public DbSet<Product> Products { get; set; } }</pre>
(B)	<pre>public class MyDataContext : DbContext { public DbSet<Customer> Customers { get; set; } public DbSet<Product> Products { get; set; } }</pre>
(C)	<pre>public class MyDataContext : DbContext { public DbSet<Customer> Customers { get; set; } public DbSet<Product> Products { get; set; } }</pre>

(D)	public class MyDataContext : DbContext { public DbSet<Customer> Customers { get; set; } public DbSet<Product> Products { get; set; } }
-----	--

(A)	B	(C)	D
(B)	C	(D)	A

6.6.1 Answers

(1)	D
(2)	A
(3)	D
(4)	C
(5)	B



Summary

- The Entity Framework is an ORM framework that ASP.NET MVC applications can use.
- In the database-first approach, the Entity Framework creates model classes and properties corresponding to the existing database objects.
- In the code-first approach, the Entity Framework creates database objects based on model classes that you create to represent application data.
- The System.Data.Entity namespace of the ASP.NET MVC Framework provides a DbContext class that coordinates with Entity Framework and allows you to query and save the data in the database.
- LINQ is a set of APIs that allows you to create data-source-independent queries to implement data access in an application.
- LINQ queries operate on objects that implement either the `IEnumerable<T>` or `IQueryable<T>` interface.
- LINQ queries can also be created as method-based queries where you can directly make method calls to the standard query operator, passing lambda expressions as the parameters.

Session - 7

Consistent Styles and Layouts

Welcome to the Session, **Consistent Styles and Layouts**.

In an ASP.NET MVC Web application, you can maintain a consistent look and feel across the views. You can use the default layout file of the application to define the look and feel for the views. In addition, the MVC Framework also allows you to create your custom layouts to define styles. Apart from using the layouts in an application, you can also use the Site.css file to define styles for a particular view or for all the views available in the application.

In an ASP.NET MVC Web application, you can also create and apply adaptive styles. Applying adaptive styles allow accessing and displaying Web application in various devices irrespective of their screen size and browser capabilities.

In this Session, you will learn to:

- ➔ Define and describe how to use layout to maintain consistent look and feel
- ➔ Explain the process of creating a custom layout
- ➔ Explain how to implement styles in an application
- ➔ Explain and describe how to create adaptive styles

7.1 Consistent Look and Feel Using Layouts

Consider the scenario, where you are developing an ASP.NET MVC Web application. In this application, you have to add different views, where the Home page view contains a navigation bar which includes links that allow users to navigate through the other views of the application. Now, you want to display the navigation bar containing links on each of the views and allow easy navigation to other views in the application. In addition, you want that the navigation bar should appear in the same place on each of the views of the application.

To maintain such look and feel, you can copy the navigation bar containing links on each view manually. While doing this, you should ensure that the navigation bar is placed exactly at the same place in each of the views in the application. Whenever any changes are made on the navigation bar, it should be also applied to all the other views of the application. This process of maintaining a consistent look and feel is very complex and time consuming. To overcome such problems, ASP.NET MVC Framework allows you to use layouts that simplify the process of maintaining consistent look and feel in an application.

You can use layout that allows you to specify a style template to be used across the views in an application. In this layout, you can define the structure and common styles for the views.

7.1.1 _Layout.cshtml File

When you create an ASP.NET MVC Web application in Visual Studio 2013, a `_Layout.cshtml` file is automatically added in the View/Shared folder. This file represents the layout that you can apply to the views of the application. This `_Layout.cshtml` file contains the basic structure of a view.

Code Snippet 1 shows the content of the default `_Layout.cshtml` file of an application.

Code Snippet 1:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>@ViewBag.Title - My ASP.NET Application</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
<div class="navbar navbar-inverse navbar-fixed-top">
<div class="container">
<div class="navbar-header">
```

```
<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">

<span class="icon-bar"></span>
<span class="icon-bar"></span>
<span class="icon-bar"></span>
</button>

    @Html.ActionLink("Application name", "Index", "Home", null, new { @class = "navbar-brand" })

</div>

<div class="navbar-collapse collapse">
<ul class="nav navbar-nav">
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>

    @Html.Partial("_LoginPartial")

</div>
</div>
</div>

<div class="container body-content">

    @RenderBody()

<hr />

<footer>
<p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
</footer>

</div>

@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")
@RenderSection("scripts", required: false)

</body>
```

```
</html>
```

where,

The `@RenderBody()` and `@RenderSection()` methods are the placeholders for the page-specific content on each view.

The `@Styles.Render("~/Content/css")` statement renders the `.css` files.

The `@Scripts.Render("~/bundles/modernizr")` statement renders the script files in the `modernizr` library.

The `@Scripts.Render("~/bundles/jquery")` statement renders the `jQuery` library.

While developing an application, you can also modify the `_Layout.cshtml` based on your requirements. For example, you can modify the layout color, add or remove a hyperlink, and so on.

7.1.2 Custom Layout

While developing an ASP.NET MVC Web application, sometimes, you might require creating your custom layout. You can create a custom layout using the Visual Studio 2013. Creating a layout in Visual Studio 2013 is similar to creating a view. By default, the name of the layout file is preceded by the underscore (`_`) character. The extension of a layout file is `.cshtml`.

To create a custom layout using Visual Studio 2013, you need to perform the following steps:

1. Right-click the **Shared** folder under the **Views** folder in the **Solution Explorer** window.
2. Select **Add→New Item** from the context menu that appears. The **Add New Item** dialog box appears.
3. Select **MVC 5 Layout Page (Razor)** template.
4. Enter `_CustomLayout` in the **Name** text field.

Figure 7.1 shows the **Add New Item** dialog box.

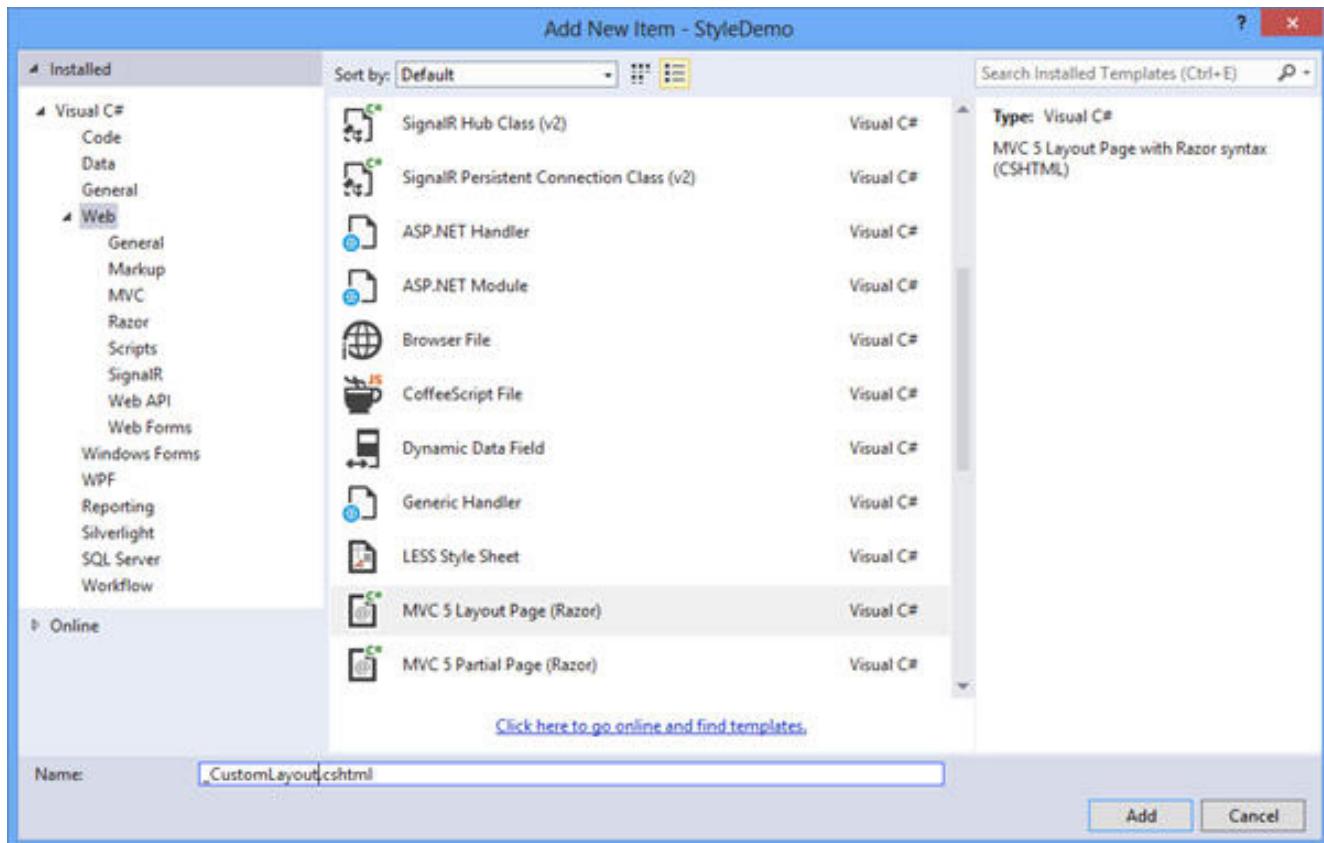


Figure 7.1: Add New Item Dialog Box

5. Click **Add**. This will add the newly created layout in the Views/Shared folder.

Figure 7.2 shows the newly created layout in the Solution Explorer window.

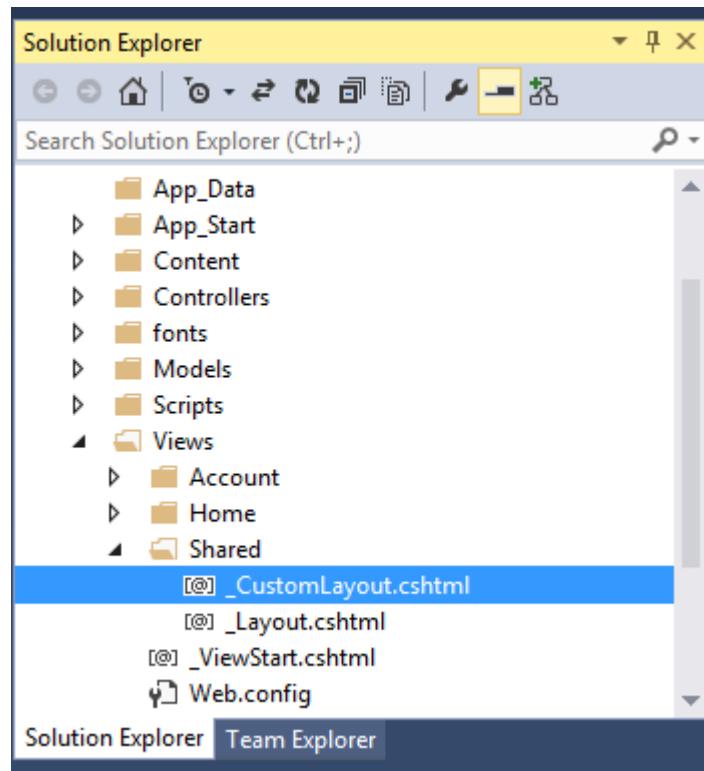


Figure 7.2: Solution Explorer Window

You can display unique content associated with a view using the layout. For this, you need to use the following methods:

- **@RenderBody():** Allows you to define a placeholder in the layout to insert the main content of the view.
- **@RenderSection():** Allows you to display a section specified in the view.

→ **@RenderBody() Method**

You can use this method to define a placeholder in the layout that needs to be inserted in the main content of the view. While using the `@RenderBody()` method, you should remember that, you can use it only once in a layout.

Code Snippet 2 shows a layout named `_CustomLayout.cshtml` file that uses the `@RenderBody()` method.

Code Snippet 2:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<title>ASP.NET MVC Application</title>
</head>
<body>
<p>
    @RenderBody()
</p>
</body>
</html>
```

In this code, the `@RenderBody()` method inside the `<p>` tag inserts the content of the view.

→ **@RenderSection() Method**

You can use this method to display a section specified in the view. A layout allows you to add one or more sections in it.

Code Snippet 3 shows the `_CustomLayout.cshtml` layout that uses the `@RenderSection()` method.

Code Snippet 3:

```
<!DOCTYPE html>
<html>
<head>
<title>@ViewBag.Title</title>
</head>
<body>
<div id="main-content">@RenderBody()</div>
<div>@RenderSection("TestSection")</div>
</body>
</html>
```

In this code, the `@RenderSection()` method is used to add a section.

Sometimes, you may not want to use a specified section in some of the views in your application. In such situation, you can use an overloaded method of the `@RenderSection()` method in your layout. This overloaded method uses the required attribute with the `false` value.

Code Snippet 4 shows how to use an overloaded method of the `@RenderSection()` method.

Code Snippet 4:

```
<div>@RenderSection("TestSection", required: false)</div>
```

In this code, the `@RenderSection()` method indicates that a view that uses this layout can provide content for the `TestSection` section. As this is not mandatory, a view using this layout may not contain any section named `TestSection`.

7.1.3 Specifying a Layout for a View

While developing an ASP.NET MVC Web application, you can also specify a layout for either a particular view or for all the views. You can specify a layout for a view by using the `Layout` property.

Code Snippet 5 shows specifying a layout for a view.

Code Snippet 5:

```
@{  
    Layout = "~/Views/Shared/CustomLayout.cshtml";  
}
```

In this code, the `CustomLayout.cshtml` file is specified as the layout for a view available in the application.

You can also specify a particular layout for all the views available in your application by using the `Layout` property in the `_ViewStart.cshtml` file. This file is available under the **Views** folder.

When you create an ASP.NET MVC Web application in Visual Studio 2013, the `_ViewStart.cshtml` file is created by default under the **Views** folder. This file specifies the default layout to be used for the views available.

Code Snippet 6 shows the default layout in the `_ViewStart.cshtml` file.

Code Snippet 6:

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}
```

In this code, the default layout is specified as `_Layout.cshtml`.

When you run the application, the code available in the `_ViewStart.cshtml` file is executed first. Then, the code of the available views in the same directory or within a sub-directory is executed. This applies the specified layout in the `_ViewStart.cshtml` file to all the views in the same directory or sub-directories.

7.1.4 Nested Layout

You have already learned that layouts are used to maintain a consistent look and feel in an application. However, while developing Web applications, sometimes, you might need to maintain some standard features. In such scenarios, you can use nested layouts.

Consider a scenario where you are developing an ASP.NET MVC Web application. In the application, you are using a layout to display a navigation bar on each view. However, you have a requirement to display a side bar in addition to the navigation bar in one or two views of your application. You can achieve this by using a nested layout.

A nested layout refers to a layout that is derived from a parent layout and is responsible for defining the structure of a page. A nested layout page uses the `@RenderBody()` method only once. It can have multiple sections that can be rendered by using the `@RenderSection()` method.

7.2 Implementing Styles

Besides using the layouts, in an ASP.NET Web application, you can also use styles that allow you to apply consistent formatting across all the views available. Styles are used to define a set of formatting options, which can be reused to format different HTML helpers on a single view or on multiple views.

7.2.1 CSS Files

A `.css` file in an ASP.NET Web application allows you to define a style that needs to be applied in the application. Whenever you create an ASP.NET MVC Web application in Visual Studio 2013, a `.css` file named `Site.css` is automatically created under the **Content** folder. You can use the `Site.css` file to include multiple style definitions.

Code Snippet 7 shows using style definitions in the `Site.css` file.

Code Snippet 7:

```
h2
{
    font-weight: bold;
    font-size: medium;
    color: red;
    font-family: Times New Roman;
}
```

This code contains a style definition that will be applied to all the `<h2>` tags in the views that references the `test.css` file. Here, `<h2>` tag acts as a selector that specifies the elements on which the specified style needs to be applied.

You can also define same style for more than one element. For this, you need to combine the selectors as a single group. For example, you can display the font of both the `<h1>` and `<h2>` elements in blue color.

Code Snippet 8 shows combining two elements in the `Site.css` file.

Code Snippet 8:

```
H1, H2 {COLOR: BLUE; }
```

This code will display the entire text that is specified inside the `<h1>` and `<h2>` elements in blue color.

Besides applying styles using elements, you can also use the following types of CSS selectors:

- **Id selector:** Is used to identify an element that you need to style differently from the rest of the page. Each element on a view can have a unique Id, which contains a hash symbol (#) followed by the element Id.

Code Snippet 9 specifies an id for the `<p>` tag.

Code Snippet 9:

```
<p id="uname">Mobile Phone Users</p>
```

Now, you can define style separately for the element, whose Id is `uname`.

Code Snippet 10 shows defining style specifications to be applied to the element, whose Id is `uname`.

Code Snippet 10:

```
#uname
{
color:green;
font-size:20pt;
font-weight:bold;
}
```

This code selects the element with the id named, `uname` and applies the specified styles on it.

- **Class selector:** Is used to specify styles for a group of elements. Unlike Id selector, you can use a class selector to apply a style on several elements in a view. For example, you have a view that contains five paragraphs. Now, you need to apply same style for the first two paragraphs. Then, the next two paragraphs will have different style. Again, the last paragraph will have another style different from the previous paragraphs. For this, you can use the class selector to define the required styles for multiple elements with the same class.

Code Snippet 11 shows how to use class selector to define styles.

Code Snippet 11:

```
<p class="red">First paragraph</p>
<p class="red">Second paragraph</p>
<p class="green">Third paragraph</p>
<p class="green">Fourth paragraph</p>
<p class="blue">Fifth paragraph</p>
```

This code uses the class selector to define styles for multiple elements.

Code Snippet 12 shows applying different styles to the five paragraphs.

Code Snippet 12:

```
.red
{
color:red;
}

.green
{
color:green;
}

.blue
{
color:blue
}
```

This code will display the first two paragraphs in red color, the next two paragraphs in green color, and the last paragraph in blue color.

Figure 7.3 shows the output of applying different styles to the five paragraphs.

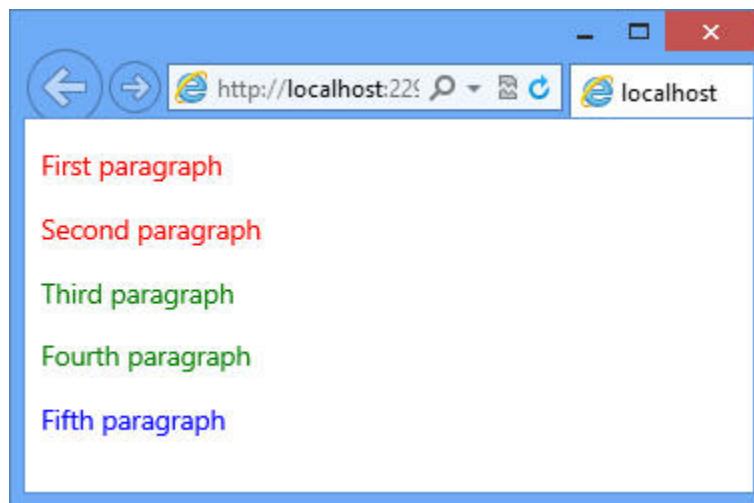


Figure 7.3: Applying Styles Using Class Selector

- **Universal selector:** Is used to specify styles for all available elements in a particular area of a page or the whole page. To define universal selector, you should use the (*) symbol. Consider a scenario, where you need to specify styles for all the elements inside the <div> tag.

Code Snippet 13 shows the <div> tag that contains some elements.

Code Snippet 13:

```
<p class="red">First paragraph</p>
<p class="red">Second paragraph</p>
<p class="green">Third paragraph</p>
<p class="green">Fourth paragraph</p>
<p class="blue">Fifth paragraph</p>
```

This code creates a <div> tag that contains three <p> tags with data inside it.

Now, to apply styles to all the elements in the preceding <div> tag, you can use the universal selector.

Code Snippet 14 shows applying styles using the universal selector.

Code Snippet 14:

```
div.Customers *
{
  color:red;
}
```

This code defines styles using the universal selector in the Site.css file.

Figure 7.4 shows the output of applying styles using the universal selector.

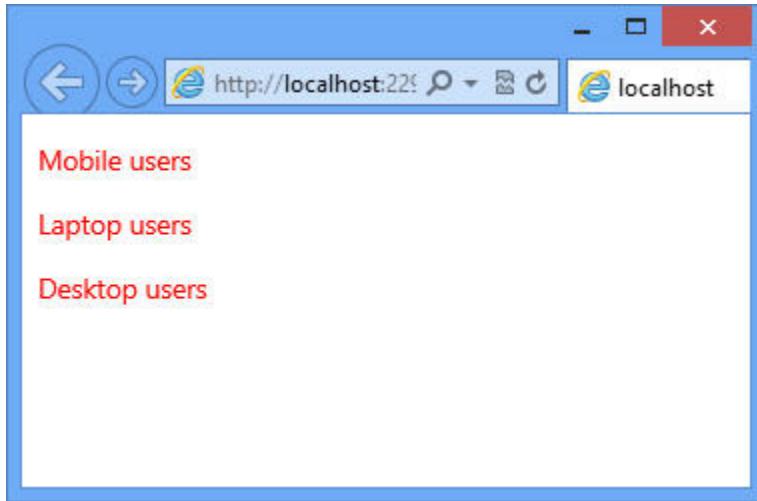


Figure 7.4: Applying Styles Using Universal Selector

7.2.2 Importing Styles in Views

Once you have defined styles in the `Site.css` file, you need to link it with HTML documents on which the styles are to be implemented. You can achieve this by using the `<link>` tag. This tag enables linking the views along with style sheet together.

Code Snippet 15 shows how to use the `<link>` tag.

Code Snippet 15:

```
<link href="~/content/Site.css" rel="stylesheet" type="text/css" />
```

In this code, the `Site.css` file is referred to by using the `href` attribute of the `<link>` tag. The value of the `rel` attribute specifies that the document will use a style sheet. Finally, the value of the `type` attribute specifies that the MIME type of the linked document is `text/css`.

7.2.3 Adaptive User Interface

With the growing technologies, at recent time, it is possible to access Web application from various devices, such as tabs, laptops, or mobile phones. As you know, that each of these devices has different screen sizes and browser capability. As a result, the page size and visual appearance of a Web application may change accordingly. Otherwise, it can hamper the functionality of the application and the user's experience.

As a solution, you can implement adaptive rendering. Adaptive rendering enables automatic scaling down of a page and redrawing the page according to a device screen.

You can create an adaptive UI in order to implement adaptive rendering.

For this, you can use one of the following mechanisms:

- Viewport meta tag
- CSS media queries

→ **Viewport Meta Tag**

The browsers of a mobile device can render pages of a Web application and scale them in a manner so that they can properly appear in the screen area of the device. These pages are rendered in a virtual window known as a viewport. Viewport is typically wider than the actual width of a mobile device.

You can also control the size and scaling of pages of a Web application. To do this, you can use the `viewport <meta>` tag in the head section of the view.

Code Snippet 16 shows how to use the `viewport <meta>` tag.

Code Snippet 16:

```
<meta name="viewport" content="width=250">
```

In this code, the `width` property is set to 250 pixels. As a result, the pages of the Web application will appear on a canvas of width 250 pixels. It will also be scaled properly inside the screen area.

You can also use the `device-width` keyword to set the width of a viewport to the native screen size of the browser. This allows you to create the content, which can be adjusted according to the device specifications.

Code Snippet 17 shows how to use the `device-width` keyword.

Code Snippet 17:

```
<meta name="viewport" content="width=device-width">
```

In this code, the `width` property is set to the `device-width` value. This specifies the width of the device screen.

→ **CSS Media Queries**

You can use CSS media queries to enable your application to support different browsers and devices. To use CSS media queries, you need to use the `@media` keyword. Using these queries, you can apply different conditional CSS styles to support different device conditions or browser capabilities.

Code Snippet 18 shows the content of an `Index.cshtml` view.

Code Snippet 18:

```
<html>
<head>
<title>Sample Article</title>
<link href="~/content/Site.css" rel="stylesheet" type="text/css" />
```

```

</head>
<body>
<div id="container">
<div id="article1">
<h1>Cricket</h1>Cricket is a bat-and-ball game played between two teams of
11 players each on a field at the center of which is a rectangular 22-yard long
pitch. Each team takes its turn to bat, attempting to score runs, while the
other team fields. Each turn is known as an innings. There are several formats of
the cricket game.
</div>
<div></div>
<div></div>
<div id="article2">
<h1>Football</h1>Football refers to a number of sports that involve, to
varying degrees, kicking a ball with the foot to score a goal. The most popular
of these sports worldwide is association football, more commonly known as
just "football" or "soccer". The variations of football are known as football
codes.
</div>
</div>
</body>
</html>

```

This code creates the `Index.cshtml` view that contains sample articles on two topics.

Once you have created the `Index.cshtml` view, then, you can apply styles in the `Site.css` file to design the `Index.cshtml` view.

Code Snippet 19 shows applying styles to design the `Index.cshtml` view.

Code Snippet 19:

```

h1 {
    text-align: center;
    color: blue;
}
#container {
    float: left;
    width: 830px;

```

```

}

#article1 {
    float: right;
    width: 410px;
    background: lightblue;
}

#article2 {
    float: left;
    width: 410px;
    background: lightblue;
}

```

This code applies styles in the `Site.css` file to design the look and feel of the `Index.cshtml` view.

Figure 7.5 shows the `Index.cshtml` view when access it on a browser.

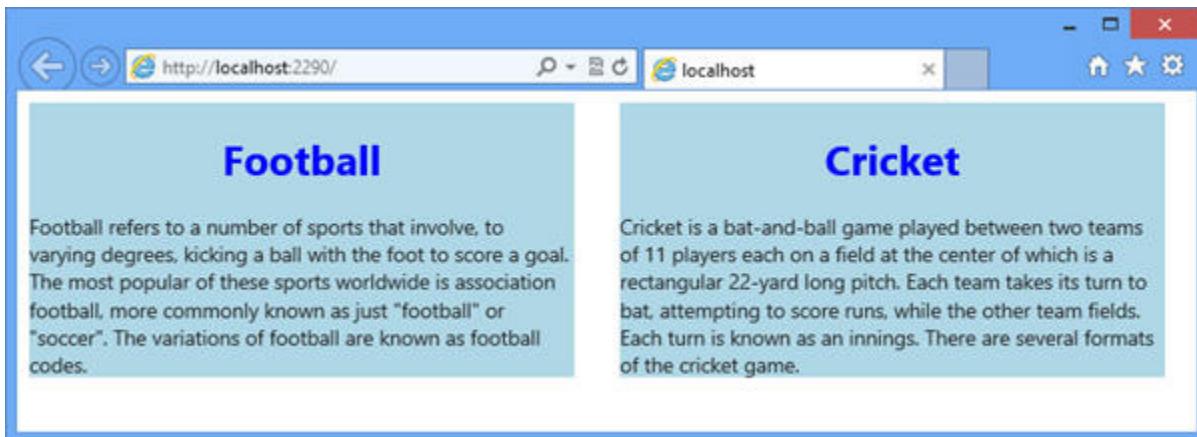


Figure 7.5: Index.cshtml View

Now, when you reduce the dimensions of the browser window, the look and feel of the view is affected.

To resolve such issues, you can use media queries in the `Site.css` file of the application to add styles that allow reducing the dimensions of a view.

Code Snippet 20 shows how to use media queries.

Code Snippet 20:

```
@media screen and (max-width:700px) {  
    #container {  
        float: left;  
        width: 500px;  
    }  
  
    #article1 {  
        float: left;  
        width: 500px;  
        background: lightblue;  
    }  
  
    #article2 {  
        float: none;  
        width: 500px;  
        background: lightblue;  
    }  
}  
  
@media screen and (max-width:500px) {  
    #container {  
        float: left;  
        width: 400px;  
    }  
  
    #article1 {  
        float: left;  
        width: 400px;  
        background: lightblue;  
    }  
  
    #article2 {  
        float: none;  
    }  
}
```

```
width: 400px;  
background: lightblue;  
}  
}
```

This code defines two media queries by using the @media syntax. The first query defines the styles for the screens that have maximum width of 700px. The second query defines the style for the screens that have maximum width of 500px. Both of these queries use the max-width property to define the styles for the screens.

When you reduce the width of the browser, the view will appear according to the styles specified with reduced screen width.

Figure 7.6 shows the `Index.cshtml` view with reduced screen width.

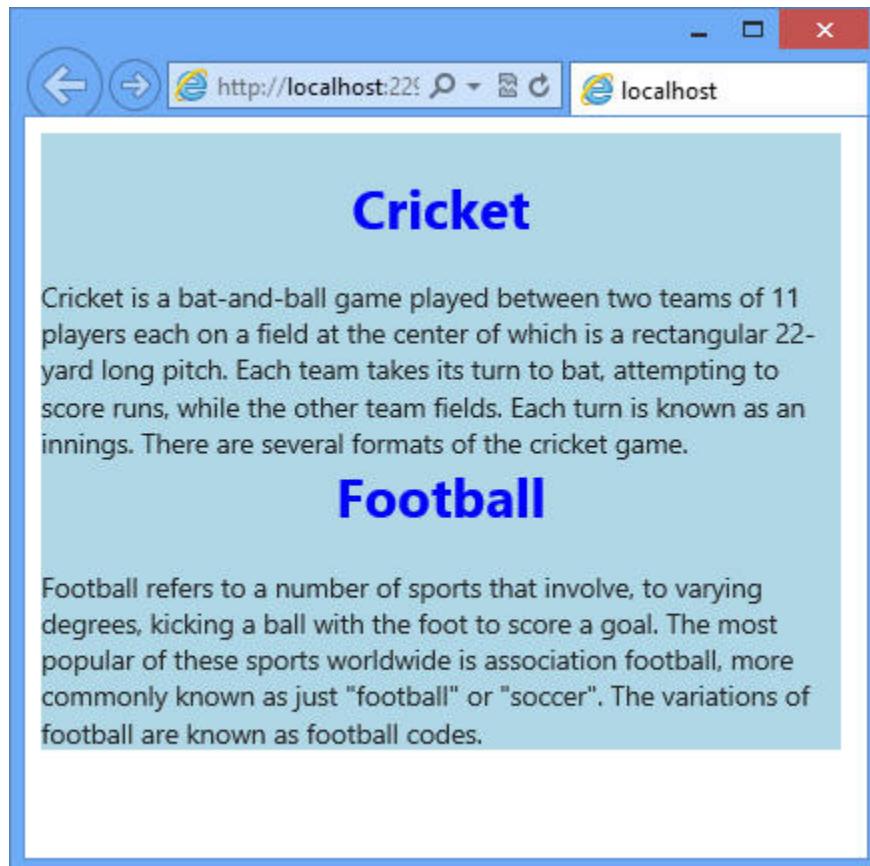


Figure 7.6: Index.cshtml View with Reduced Screen Width

You can use various properties with a media query.

Table 7.1 describes some of the common properties that you can use with a media query.

Properties	Description
Width	Specifies the width of the display area that represents the browser window.
Height	Specifies the height of the display area.
device-width	Specifies the width of the device screen.
device-height	Specifies the height of the device screen.
aspect-ratio	Specifies the ratio of the width and height properties.

Table 7.1: Properties of Media Query

7.3 Check Your Progress

1. Which of the following statements about the `_Layout.cshtml` file are true?

(A)	A <code>_Layout.cshtml</code> file defines styles for a particular or for all the views available in an application.
(B)	A <code>_Layout.cshtml</code> file allows you to create and apply adaptive styles.
(C)	A <code>_Layout.cshtml</code> file starts validation when a user enters data in a form.
(D)	A <code>_Layout.cshtml</code> file contains the basic structure of a view.
(E)	A <code>_Layout.cshtml</code> file is the default layout of an application stored in the View/Shared folder.

(A)	A, B	(C)	A, D
(B)	C, D	(D)	D, E

2. Which of the following file contains the code that is executed first, when you run an application?

(A)	packages.config
(B)	web.config
(C)	Site.css
(D)	<code>_ViewStart.cshtml</code>
(E)	<code>_Layout.cshtml</code>

(A)	B	(C)	D
(B)	C	(D)	A

3. Which of the following options will you use to define a placeholder in the layout to insert the main content of the view?

(A)	<code>@RenderSection()</code>
(B)	<code>@RenderBody()</code>
(C)	<code>@Html.LabelFor()</code>
(D)	<code>@Scripts.Render()</code>
(E)	<code>@import statement</code>

(A)	B	(C)	D
(B)	C	(D)	A

4. Which of the following code snippets correctly uses the `<link>` tag to link the default .css file?

(A)	<code><link href="~/content/Site.css" rel="stylesheet" type="text/css" /></code>
(B)	<code><link href="~/views/Site.css" rel="text/css" type="stylesheet"/></code>
(C)	<code><link =("~/content/Site.css" rel="stylesheet" type="text/css"/></code>
(D)	<code><link href="~/content/stylesheets" rel="Site.css" type="text/css" /></code>
(E)	<code><link href="~/views/Site.css" rel="stylesheet" type="text/css"/></code>

(A)	B	(C)	D
(B)	C	(D)	A

5. Which of the following code snippets correctly defines a style specification to be applied on the text inside the `<p>` element?

Assuming that you have the following code that specifies an id for the `<p>` tag:

```
<p id="uname">Mobile Phone Users</p>
```

(A)	<pre>#uname { color:green; font-size:20pt; font-weight:bold; }</pre>
(B)	<pre>@p { color:green; font-size:20pt; font-weight:bold; }</pre>

(C)	*uname { Color.green; font-size.20pt; font-weight.bold; }
(D)	#uname { color:green; font-size:20pt; font-weight:bold; }

(A)	B	(C)	D
(B)	C	(D)	A

7.3.1 Answers

(1)	D
(2)	C
(3)	B
(4)	A
(5)	C



Summary

- ASP.NET MVC Framework allows you to use layouts to simplify the process of maintaining consistent look and feel in an application.
- The _Layout.cshtml file represents the layout that you can apply to the views of an application.
- You can specify a layout for a view by using the Layout property.
- A nested layout refers to a layout that is derived from a parent layout and is responsible for defining the structure of a page.
- Styles are used to define a set of formatting options, which can be reused to format different HTML helpers on a single view or on multiple views.
- The Site.css file allows you to define styles for a particular view or for all the views available in the application.
- Adaptive rendering enables automatic scaling down of a page and redrawing the page according to a device screen.

Session - 8

Responsive Pages

Welcome to the Session, **Responsive Pages**.

An ASP.NET MVC Web application should be responsive. A responsive Web application is a dynamic and interactive application that enhances user experience. Such responsive Web applications can be created using JavaScript, which is a client-side scripting language that enables a Web application to respond to user requests without interacting with a Web server.

In addition, you can also use jQuery in your Web application to make your application responsive. jQuery is a JavaScript library that you can use to apply various effects to the various UI elements in your application.

Another approach to make an ASP.NET MVC application responsive is to use Asynchronous JavaScript and XML (AJAX), which is a Web development technique to create dynamically interactive applications.

In this Session, you will learn to:

- ➔ Define and describe how to use JavaScript
- ➔ Define and describe how to use jQuery
- ➔ Define and describe AJAX
- ➔ Explain and describe how work with AJAX

8.1 Using JavaScript

A dynamic and interactive ASP.NET MVC application must implement functionalities, such as an easy to use UI, quick response to the user's request. In addition it should run in all available browsers.

To achieve this in an ASP.NET MVC application, you can use JavaScript. JavaScript is a client-side scripting language that allows you to develop dynamic and interactive Web applications. When you use JavaScript, your Web applications can respond to user requests without interacting with a Web server. As a result, it reduces the response time to deliver Web pages faster.

Code Snippet 1 shows a JavaScript in an HTML page.

Code Snippet 1:

```
<!DOCTYPE html>
<html>
<body>
<p>Click the Alert Box button.</p>
<button onclick="scriptFunction()">Alert Box </button>
<script>
function scriptFunction()
{
    alert("Welcome user")
}
</script>
</body>
</html>
```

This code uses JavaScript to display a view that contains an **Alert Box** button.

Figure 8.1 shows the `Index.cshtml` view that displays the **Alert Box** button.

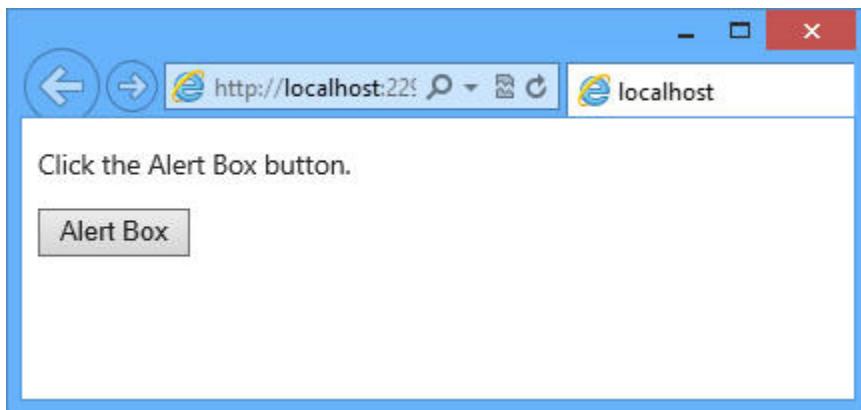


Figure 8.1: Output of the `Index.cshtml` View

When a user clicks the **Alert Box** button, the JavaScript code will execute on the browser and display an alert box.

Figure 8.2 shows the alert box that appears when the user clicks the Alert Box button.

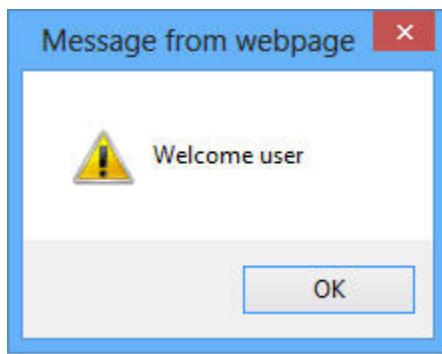


Figure 8.2: Displaying Alert Box

8.1.1 Unobtrusive JavaScript

Traditionally in a Web application, JavaScript is used in the same file with HTML code. However, using JavaScript with HTML code is not recommended. This is because, in a Web page, you should always separate the presentation from the business logic code. As a solution, you can use unobtrusive JavaScript to separate the JavaScript code from HTML markups.

Unobtrusive JavaScript is an approach to use JavaScript separately from HTML markups in Web applications. You can store the JavaScript code in a file with the `.js` extension.

Note - As convention, you should store this `.js` file in the Script folder of the application directory in an ASP.NET MVC application.

When you use Visual Studio 2013 to create an ASP.NET MVC application, it automatically creates a Scripts folder that contains various JavaScript files.

Figure 8.3 shows the Solution Explorer window that displays the .js files in the Scripts folder.

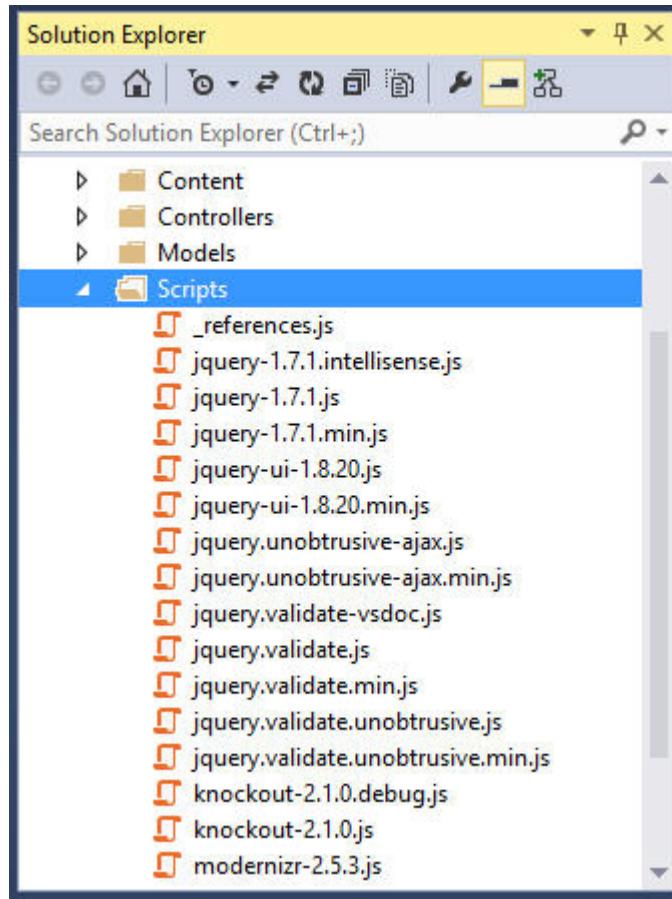


Figure 8.3: Solution Explorer Window

Once you have a JavaScript file with .js file extension, you can refer it in a view. To do this, you need to use the <script> element.

The syntax of using the <script> element in a view is as follows:

Syntax:

```
<script type=<script_type>src=<script_source>>
</script>
```

where,

script_type: Specifies the type of scripting that has to be used, for example “text/javascript”

script_source: Specifies the source of the jQuery library that has to be used, for example “@Url.Content(“~/Scripts/TestScripts.js”)”

Code Snippet 2 shows using the `<script>` element.

Code Snippet 2:

```
<script src="@Url.Content("~/Scripts/Testscript.js")" type="text/javascript">
</script>
```

This code uses the `<script>` element to refer a JavaScript file named `TestScripts.j`.

When you create an ASP.NET MVC application in Visual Studio 2013, a Script folder is automatically created with several JavaScript libraries. You can refer one of those libraries in a view and then, use the functionalities provided by the library.

The default JavaScript library files that Visual Studio 2013 provides two versions. One version is the main JavaScript library and another version is the minified version of the main JavaScript library. The minified version contains the word `min` in their name. For example, the `jquery-1.7.1.js` main JavaScript library has an accompanying `jquery-1.7.1.min.js` library.

The minified versions are smaller in size compared to their corresponding main version as they do not contain unnecessary characters, such as white spaces, new lines, and comments. However, the minified versions provide the same functionalities as their corresponding main versions. The advantage of using the minified versions is that it helps reduce the amount of data to be packaged in an application.

The Visual Studio 2013 automatically generates the following characteristics:

- Files with the `vsdoc` word in their name, are annotated to help Visual Studio to provide better intellisense.
- Files with the `modernizr` word in their name enable you to take the advantages of the evolving Web technologies.
- Files with the `knockout` word in their name are responsible for providing data binding capabilities.

8.2 Introducing jQuery

jQuery is a cross-browser JavaScript library that you can use to perform various functionalities, such as finding, traversing, and manipulating HTML elements. Apart from these functionalities, jQuery also enables you to animate HTML elements, handle events to make views of an application more dynamic and interactive.

You can make use of the jQuery library in a view of your application by using the `<script>` element in the head section of the view.

Code Snippet 3 shows using the `<script>` element.

Code Snippet 3:

```
<script type="text/javascript">
    src="@Url.Content("~/Scripts/jquery-1.7.1.js")"
</script>
```

This code uses the `<script>` element to use jQuery library in a view.

jQuery provides the `document.ready()` function to ensure that a view of the Web application has been fully loaded before executing the jQuery code. Consider a scenario, where you have added a jQuery code to bounce an image. If the code for bouncing the image executes before the document is ready, it results an error. This is because, the script will try to bounce the image that is not yet loaded in the view. To avoid such problems, you should write the code that needs to be executed after a page loads inside the `(document).ready()` function.

The syntax of using the `document.ready()` function is as follows:

Syntax:

```
$ (document).ready(function()
{
    // jQuery code...
});
```

where,

`($)`: Represents the start of a jQuery code

`(document).ready()`: Checks the readiness of the actions performed on an HTML element

Sometimes, you might require adding customized client-side functionality in your application. In such situation, you can add your jQuery code in a file with `.js` extension and then, add it to the Scripts folder of the application directory.

Visual Studio 2013 simplifies the process of creating a JavaScript file that can contain jQuery code for Web applications. To create a JavaScript file in Visual Studio 2013, you need to perform the following tasks:

1. Right-click the **Scripts** folder in the **Solution Explorer** window.
2. Select **Add→New Item** from the context menu that appears. The **Add New Item** dialog box appears.
3. Select **JavaScript File** template in the **Add New Item** dialog box.

Type `TestScripts` in the **Name** text field.

Figure 8.4 shows the **Add New Item** dialog box.

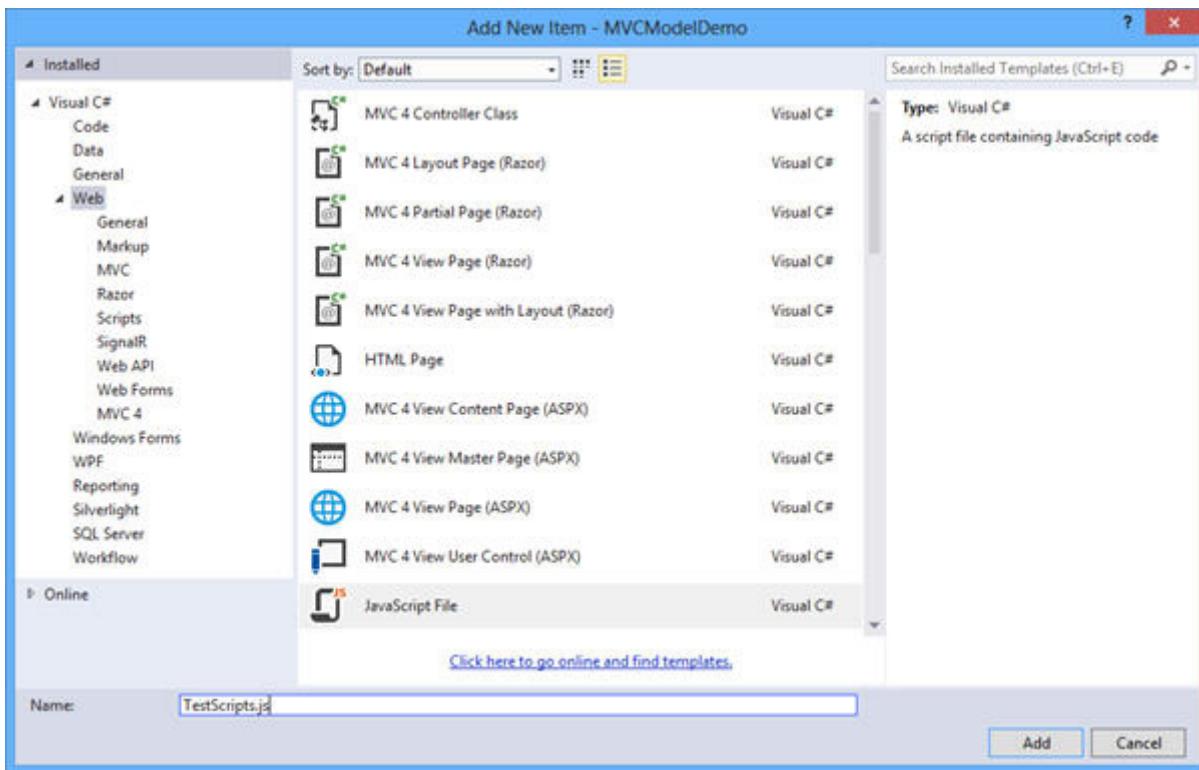


Figure 8.4: Add New Item Dialog Box

- Click **Add**. This will add the newly created `TestScripts.js` file in the **Scripts** folder.

Once you have created and added the jQuery code in a `.js` file, you need to invoke it. For that you need to refer to the `.js` file by using the `<script>` element.

Code Snippet 4 shows referring to a customized `TestScripts.js` file.

Code Snippet 4:

```
<script type="text/javascript"
src("~/Scripts/TestScripts.js"></script>
```

This code uses the `<script>` element that refers to the `TestScripts.js` file.

Apart from this, you can also use jQuery to select and manipulate HTML elements, handle events related to HTML elements, and add effects to HTML elements.

→ Selecting and Manipulating HTML Elements

To select HTML elements and then, manipulating them, jQuery allows you use some specific syntax.

The syntax to select HTML elements and then, perform the required actions on them is as follows:

Syntax:

`$(selector).action()`

where,

`$`: Defines or access jQuery.

`selector`: Finds an HTML element or a group of HTML elements based on their name, id, class, and attribute.

`action()`: Specifies the action to be performed on the HTML element.

You can use one of the following jQuery selectors, as per your requirements:

- **Element selector:** Allows searching an HTML element or a group of HTML elements on the basis of their names. For example, to search all the h2 elements, you can use `$(“h2”)`.
- **ID selector:** Allows searching elements on the basis of their ids. For example, to search an element with the id, green, you can use `$(“#green”)`.
- **Class selector:** Allows searching elements on the basis of their class names. For example, to search all elements with the class, named red you can use `$(“.red”)`
- **Handling Events**

In jQuery, to handle events you can use functions or built-in event methods. You can use an event method in order to select an event and then, trigger a function when that event occurs. jQuery provides various event methods that you can use to handle events on HTML element.

Table 8.1 describes some of the event methods of jQuery.

Event Method	Description
<code>\$(document).ready(function)</code>	Allows executing a function once a document completely loads.
<code>\$(selector).click(function)</code>	Allows executing a function on the click event of the selected elements.
<code>\$(selector).dblclick(function)</code>	Allows executing a function on the double-click event of the selected elements.
<code>\$(selector).mouseover(function)</code>	Allows executing a function when the mouse pointer is moved over the selected elements.
<code>\$(selector).mouseout(function)</code>	Allows executing a function when the mouse pointer is moved out of the selected elements.
<code>\$(selector).keydown(function)</code>	Allows executing a function when a key is pressed on the selected elements.

Table8.1: Built-in Event Methods of jQuery

Consider a scenario where you want to use a jQuery code to change the text inside the `<h1>` element. When a user clicks the **Click to see the changes** button the text inside the `<h1>` element should be changed in the view.

Code Snippet 5 shows the code of the `Index.cshtml` view.

Code Snippet 5:

```
<html>
<head>
<title>Using jQuery</title>
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery-1.7.1.js")"></script>
<script src="@Url.Content("~/Scripts/Testscripts.js")" type="text/javascript"></script>
</head>
<body>
<h1>This is a header.</h1>
<hr />
<input type="button" id="change_header" value='Click to see the changes' />
</body>
</html>
```

This code creates an `Index.cshtml` view that contains a button named, `Click to see the changes`, and added a reference to the script file, named `Testscripts.js`.

Once, you have created the `Index.cshtml` view, you need to add the jQuery code in the `TestScripts.js` file to change the text inside the `<h1>` element.

Code Snippet 6 shows the content of the `TestScripts.js` file.

Code Snippet 6:

```
$(document).ready(function () {
    $("#change_header").click(function () {
        $("h1").html("This header has changed.");
    });
});
```

In this code, jQuery identifies a click event when the user clicks the **Click to see the changes** button. Then, the function associated with the click event of the button is executed and the text inside the `<h1>` element changes.

Figure 8.5 shows the `Index.cshtml` view, when you access the application.

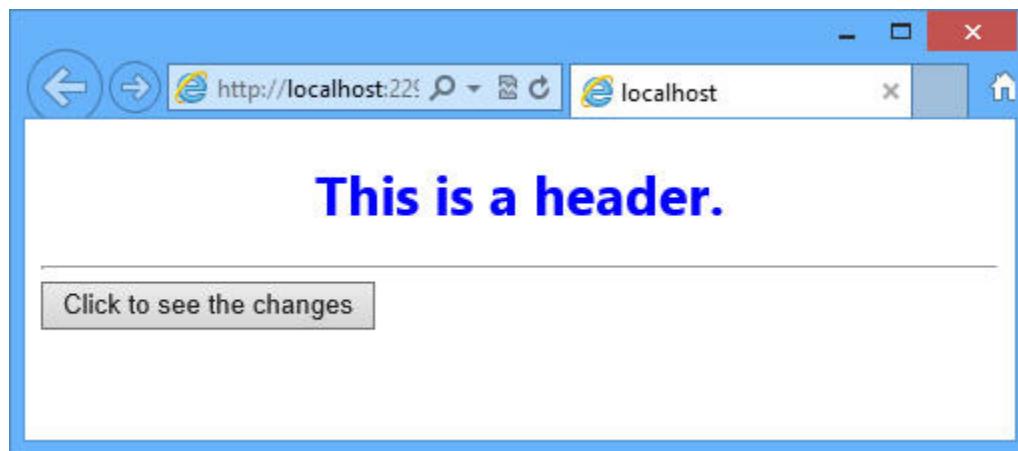


Figure 8.5: Index.cshtml View

When you click the **Click to see the changes** button the text inside the `<h1>` element that is "This is a header should be changed".

Figure 8.6 shows the changes of the text inside the `<h1>` element.

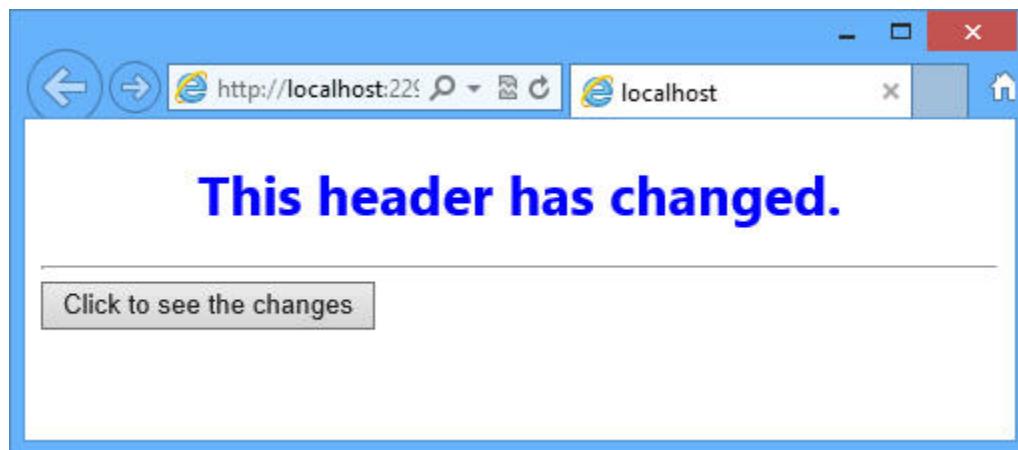


Figure 8.6: Changed Text of the `<h1>` Element

→ Adding Different Effects

Using jQuery, you can perform various functionalities on HTML elements. For this, jQuery provides various built-in actions, such as hide and fade effect which can be applied to the HTML elements. These effects enable you to enrich the browsing experience of your user.

→ Hide Effect

You can use the `hide()` function to make an element hidden when an event like click or double-click occurs.

The general syntax of using the `hide()` function to hide the selected elements.

Syntax:

```
$(selector).hide(speed)
```

where,

`speed`: Specifies the speed at which an element disappears. You can use one of the following values for the `speed` attribute:

- `slow`
- `fast`
- duration in milliseconds

When you specify the values for the `speed` attribute as duration in milliseconds, you should ensure that this value should not enclosed within quotes. On the other hand, when you specify values for the `speed` attribute as `slow` or `fast`, this value should be within the quotes.

Consider a scenario, where you want to slowly hide a text when a user clicks a button in a view. In such scenario, you can use the `hide()` function.

Code Snippet 7 shows the `Index.cshtml` view that contains a **Click to hide the text** button.

Code Snippet 7:

```
<html>
<head>
<title>Hide effect</title>
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery-1.7.1.js")"></script>
<script src="@Url.Content("~/Scripts/TestScripts.js")" type="text/javascript"></script>
</head>
<body>
<h1>Text to be disappeared</h1>
<button>Click to hide the text</button>
</body>
</html>
```

This code contains the text, where the hide effect is to be applied, and a reference to the file that contains the jQuery code with hide effects is specified.

Now, to hide the text, when a user clicks the button you need to use the `hide()` function in the `.js` file.

Code Snippet 8 shows the jQuery code in the `TestScripts.js` file to hide a text.

Code Snippet 8:

```
$ (document) .ready(function() {
    $("button") .click(function() {
        $("h1") .hide("slow");
    });
});
```

In this code, the text “Text to be disappeared” is displayed before the **Click to hide the text** button. When the user clicks this button, the text “Text to be disappeared” slowly disappears from the browser window.

→ Fade Effect

You can use the fade effect on a selected element using jQuery. This effect enables you to gradually reduce the opacity of the selected elements. Some of the common fade function that you can use on selected elements are `fadeOut()` and `fadeIn()`.

Consider a scenario, where you want to fade out the text “Text to be faded out” when a user clicks over it. Thereafter, you want to fade in and display the disappeared text when a user clicks the **Restore Text** button. In such scenario, you can use the fade effect of jQuery.

Code Snippet 9 shows the `Index.cshtml` file that contains the text where the fade effect is to be applied and a **Restore Text** button.

Code Snippet 9:

```
<html>
<head>
<title>Hide Effect</title>
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery-1.7.1.js")"></script>
<script src="@Url.Content("~/Scripts/TestScripts.js")" type="text/javascript"></script>
</head>
<body>
<p>Text to be faded out!</p>
<br />
<button>Restore Text</button>
</body>
```

```
</html>
```

This code contains the text, where the fade effect is to be applied, and a reference to the script file that contains the jQuery code that defines fade effects.

Once you have created the view, you can define the jQuery code to implement fade effect in the `TestScripts.js` file.

Code Snippet 10 shows the jQuery code to perform fade in and fade out effects.

Code Snippet 10:

```
$ (document) .ready(function () {
    $("p") .click(function () {
        $(this) .fadeOut(1400);
    });
    $("button") .click(function () {
        $("p") .fadeIn(1000);
    });
});
```

In this code, the `<p>` element is referred using its name. The `this` keyword in the click event handler of the `<p>` element refers to the current HTML element, which is the `<p>` element. When the user clicks the text 'Text to be faded out' the text fades away. When the user clicks the **Restore Text** button, the text "Text to be faded out" reappears.

8.2.1 JQuery UI

JQuery UI is set of features that allows you to develop highly interactive Web applications. It contains interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript library.

JQuery UI library provides a set of widgets that you can use to design UIs and apply various effects to the UI elements. You can also apply themes to the widgets to integrate their colors and styles according to your requirements.

You can use JQuery UI library in a view by including a reference to the JQuery UI script file in the view.

Code Snippet 11 shows adding reference to the JQuery UI script file in a view.

Code Snippet 11:

```
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery-
ui-1.8.20.js")"></script>
```

→ Applying jQuery UI Effects

You can add various effects to the various UI elements in your application. While applying the style animations, jQuery UI provides a technique that you use to add, remove, or toggle class for the HTML elements.

Code Snippet 12 shows the `Index.cshtml` view that contains three buttons.

Code Snippet 12:

```
<html>
<head>
<title>Using jQuery UI</title>
<link rel="stylesheet" href="@Url.Content("~/Content/main.css")">
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery-1.7.1.js")"></script>
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery-ui-1.8.20.js")"></script>
<script src="@Url.Content("~/Scripts/TestScripts.js")" type="text/javascript"></script>
</head>
<body>
<div id="div1"></div>
<button id="button1">Add new color</button>
<button id="button2">Remove a color</button>
<button id="button3">Toggle color</button>
</body>
</html>
```

This code creates a view that contains a `<div>` element and three buttons, named Add new color, Remove a color, and Toggle color. The view uses the default a stylesheet of the application named `Site.css`.

Code Snippet 13 shows the defined styles in the `Site.css` file.

Code Snippet 13:

```
div {
width:350px;
height: 250px;
background-color:blue;
```

```
.
myclass {
width : 350px;
height: 250px;
background-color:purple;
}
```

This code defines style to the `<div>` and `myclass` elements.

Once you define the style in the `Site.css` file, you can define jQuery code to perform these effects in the script file named, `TestScripts.js`.

Code Snippet 14 shows the `TestScripts.js` that contains jQuery code to perform these effects.

Code Snippet 14:

```
$ (document) .ready(function () {
  $("#button1") .click(function () {
    $("#div1") .addClass("myclass", 550);
  });
  $("#button2") .click(function () {
    $("#div1") .removeClass("myclass", 550);
  });
  $("#button3") .click(function () {

    $("#div1") .toggleClass("myclass", 550);

  });
});
```

This code applies different effects, such as when a user clicks the **Add new color** button, the `myclass` class defined in the `Site.css` file will be applied to the `<div>` element. Secondly, when a user clicks the **Remove a color** button, the `myclass` class that is applied to the `<div>` element will be removed. Finally, when the user clicks the **Toggle colors** button, the `myclass` class will be applied to the `<div>` element if it is not already applied otherwise the class will be removed from the `<div>` element.

8.3 Introducing AJAX

AJAX is a Web development technique that allows you to make requests to the server in the background using client-side code. Thus, it enables you to update a view without reloading it completely. As a result, it improves the performance of your application and the user experience.

To understand the concept of AJAX, you should understand about asynchronous communication. Asynchronous communication is the ability of a Web application to send multiple requests and receive responses from the server simultaneously. This enables you to work on the application without being affected by the responses received from the server.

8.3.1 AJAX Helpers

You have already learned about the HTML helpers that enable you to create forms and links that point to controller actions. Similarly, ASP.NET MVC application also uses a set of AJAX helpers. These AJAX helpers enable you to create forms and links that point to controller actions. The only difference is that AJAX helpers behave asynchronously.

While using AJAX helpers, you should not explicitly write JavaScript code make the asynchrony work. To make use of it, you should ensure that the `jquery.unobtrusive-ajax` script file is present in the `Scripts` folder of your application directory.

However, if you want to include the file manually, you need to use the `<script>` element.

Code Snippet 15 shows adding reference to the `jquery.unobtrusive-ajax` script file manually in the default `_Layout.cshtml` file.

Code Snippet 15:

```
<script src="~/Scripts/jquery-1.7.1.min.js">
</script>

<script src="~/Scripts/Scripts/jquery.unobtrusive-ajax.min.js">
</script>

@RenderSection("scripts", required:false);
```

This code adds reference to the `jquery.unobtrusive-ajax` script file in the default `_Layout.cshtml` file.

8.3.2 Unobtrusive AJAX

The ASP.NET MVC Framework provides built-in support for unobtrusive AJAX. Thus, it enables you to use AJAX helpers to define AJAX features instead of writing code in your views.

To enable unobtrusive AJAX feature in an application, you need to configure the `Web.config` file. To enable unobtrusive AJAX feature in the `Web.config` file, you need to set the `UnobtrusiveJavaScriptEnabled` property to true under the `<appSettings>` element of the `Web.config`.

Code Snippet 16 shows enabling the unobtrusive AJAX feature in the `Web.config` file.

Code Snippet 16:

```
<appSettings>
<add key="webpages:Enabled" value="false" />
<add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

This code sets the value of the `UnobtrusiveJavaScriptEnabled` property to true to enable the unobtrusive AJAX feature.

In addition to enabling the unobtrusive AJAX feature in the `Web.config` file, you need to add references to the jQuery JavaScript libraries that implement the unobtrusive AJAX functionality.

For that you need to add reference of the jQuery JavaScript libraries in the views that need to use it.

Code Snippet 17 shows adding references to jQuery JavaScript libraries in a view.

Code Snippet 17:

```
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery-
1.7.1.min.js")">
</script>
<script src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.min.js")"></
script>
```

In this code, the `jquery-1.7.1.min.js` file contains the core jQuery library, and the `jquery.unobtrusive-ajax.min.js` file contains the AJAX library.

8.4 Working with AJAX

Traditionally, to update the content of a view the full page refresh technique was used. In this technique, to update the content of a view an element in the page triggers a request to the server. When the server finishes processing the request, a new page is sent back to the browser. However, AJAX provides an approach to improve the process of updating views. In an ASP.NET MVC application, you can implement AJAX using `AJAX.ActionLink()` and `AJAX Forms`.

8.4.1 AJAX ActionLinks

Similar to the HTML helper methods, most of the methods of `AJAX` property are extension methods. The `Ajax.ActionLink()` helper method enables you to create an anchor tag with asynchronous behavior.

Consider a scenario, where you want to display an `Attachment` link at the bottom of a view. When users click the link, you want to display the image of the attachment in the existing view instead of navigating to another view. In such scenario, you can use the `Ajax.ActionLink()` method.

Code Snippet 18 shows using the `Ajax.ActionLink()` method in the `Index.cshtml` view.

Code Snippet 18:

```
<div id="attachmentImage">
    @Ajax.ActionLink("Attachment", "AttachmentImage",
        new AjaxOptions
    {
        HttpMethod = "Post",
        InsertionMode = InsertionMode.Replace,
        LoadingElementId = "loadingImage",
        UpdateTargetId = "attachmentImage"
    })
</div>

<div id="loadingImage" style="display:none">
    Loading attachment...
</div>
```

In this code, the first parameter of the `Ajax.ActionLink()` method specifies the link text, and the second parameter specifies the name of the action method that needs to be invoked asynchronously.

8.4.2 AJAX Forms

Consider a scenario where you want to enable a user to search for product. When a user types a text to search for a product, the details of the matching products should be retrieved from the server and displayed on the same page, without having to post the page back to the server. In such scenario, you should use an asynchronous form element on the view.

Code Snippet 19 shows using an asynchronous form element.

Code Snippet 19:

```
@using (Ajax.BeginForm("ProductSearch", "Product",
    new AjaxOptions {
        InsertionMode = InsertionMode.Replace,
        HttpMethod = "GET",
        OnFailure = "searchFailed",
        LoadingElementId = "ajax-loader",
        UpdateTargetId = "searchresults",
    })
)
```

```
}) )
{
<input type="text" name="q" />
<input type="Submit" value="search" />

}
```

The code creates an AJAX form that contains a text box and a **Submit** button. This form also contains a `<div>` element with the `id`, `searchresults` on the view. The first parameter of the `Ajax.BeginForm()` method specifies the `ProductSearch()` action method that handles the AJAX request on the server. The second parameter specifies the `Product` controller class that contains the action method. The third parameter specifies various options for the AJAX request.

When the user clicks the Submit button, the browser sends an asynchronous GET request to the `ProductSearch()` action method of the `Product` controller.

Code Snippet 20 shows defining the `ProductSearch()` action method.

Code Snippet 20:

```
public ActionResult ProductSearch(string q)
{
    ProductDBContext db = new ProductDBContext();
    var products = db.Products.Where(a => a.name.Contains(q));
    return PartialView("_searchresults", products);
}
```

This code retrieves product records that match the user specified search string from the database. Then, it returns a partial view named `_searchresults` that contains the retrieved records. This partial view is then, rendered in the `searchresults` element on the view.

8.5 Check Your Progress

1. Which of these statements about the Unobtrusive JavaScript are true?

(A)	Unobtrusive JavaScript is an approach that allows removing unnecessary characters, such as white spaces, new lines, and comments from views.		
(B)	Unobtrusive JavaScript is an approach that allows reducing the amount of data to be packaged in an application.		
(C)	Unobtrusive JavaScript can be used in a view of an application by using the <code><script></code> tag.		
(D)	Unobtrusive JavaScript is an approach that allows using JavaScript the same file with HTML code.		
(E)	Unobtrusive JavaScript is an approach to use JavaScript separately from HTML markups in Web applications.		

(A)	A, B	(C)	A, D
(B)	C, E	(D)	D, E

2. Which of the following files of the JavaScript libraries enables you to take the advantages of the evolving Web technologies?

(A)	File with the <code>vsdoc</code> word in their name.		
(B)	Files with the <code>modernizr</code> word in their name.		
(C)	Files with the <code>knockout</code> word in their names.		
(D)	Files with the <code>min</code> word in their names.		

(A)	B	(C)	D
(B)	C	(D)	A

3. Which of the following options ensures that a view of the Web application has been fully loaded before executing the jQuery code?

(A)	<code>document.ready()</code> function		
(B)	<code>@RenderBody()</code> method		
(C)	<code>@Url.Content</code> function		
(D)	<code>@Scripts.Render()</code> method		
(E)	<code>\$(selector).action()</code> function		

(A)	B	(C)	D
(B)	C	(D)	A

4. Which of the following event methods of jQuery allows executing a function once a document completely loads?

(A)	<code>\$(selector).keydown(function)</code>
(B)	<code>\$(selector).click(function)</code>
(C)	<code>\$(selector).mouseover(function)</code>
(D)	<code>\$(selector).mouseout(function)</code>
(E)	<code>\$(document).ready(function)</code>

(A)	B	(C)	D
(B)	E	(D)	A

5. Which of the following code adds reference to a jQuery UI script file in a view?

(A)	<code><script type="source/javascript" src="@Url.Content("~/Scripts/jquery-ui-1.8.20.js")"></script></code>
(B)	<code><script type="source/javascript" src="@Url.Content("~/Scripts/JavaScript-ui-1.8.20.min")"></script></code>
(C)	<code><script type="text/jQuery" src="@Url.Content("~/Scripts/jquery-ui-1.8.20.js")"></script></code>
(D)	<code><script type="text/javascript" src="@Url.Content("~/Scripts/jquery-ui-1.8.20.js")"></script></code>
(E)	<code><script="text/javascript" src="@Url.Content("~/Scripts/jquery-ui-1.8.20.js")"></script></code>

(A)	B	(C)	D
(B)	C	(D)	A

8.5.1 Answers

(1)	B
(2)	A
(3)	D
(4)	B
(5)	C

Summary

- JavaScript is a client-side scripting language that allows you to develop dynamic and interactive Web applications.
- Unobtrusive JavaScript is an approach to use JavaScript separately from HTML markups in Web applications.
- jQuery is a cross-browser JavaScript library that you can use to perform various functionalities, such as finding, traversing, and manipulating HTML elements.
- jQuery UI library provides a set of widgets that you can use to design UIs and apply various effects to the UI elements.
- AJAX is a Web development technique that allows you to make requests to the server in the background using client-side code.
- ASP.NET MVC application uses a set of AJAX helpers that enable you to create forms and links that point to controller actions.
- The ASP.NET MVC Framework provides built-in support for unobtrusive AJAX that enables you to use AJAX helpers to define AJAX features instead of writing code in your views.

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION



Session - 9

State Management and Optimization

Welcome to the Session, **State Management and Optimization**.

The ASP.NET MVC Framework provides state management features that automatically indicate whether the sequential requests are coming from the same or different clients. You can use state management using several options such as cookies, TempData, application state, and session state.

The ASP.NET MVC Framework also provides a feature called caching that enables you to improve the performance of your Web application.

Apart from these features, the ASP.NET MVC Framework also provides two techniques known as Bundling and Minification. You can use these techniques to improve the load time of a Web page in an application.

In this Session, you will learn to:

- ➔ Define and describe state management
- ➔ Explain and describe how to use cookies and application and session state
- ➔ Explain how to use caches to improve performance of an application
- ➔ Explain the process of bundling and minification

9.1 State Management

Web pages use HTTP protocol to communicate between a Web browser and a Web server. HTTP is a stateless protocol and cannot automatically indicate whether the sequential requests are coming from the same or different clients. To overcome such problem, ASP.NET MVC provides state management features. You can implement state management in an ASP.NET MVC application using one of the following options:

- Cookies
- TempData
- Application State
- Session State

9.1.1 Cookies

Cookies are used to store small pieces of information related to a user's computer, such as its IP address, browser type, operating system, and Web pages last visited. The purpose of storing this information is to offer a personalized experience to the user. For example, if a user has logged on to your Website for the first time, the name of the user can be stored in a cookie. This information can be retrieved later on when the same user visits the Website again to greet the user with his/her name.

Cookies are sent to a client computer along with the page output. These cookies are stored on the client's computer. When a browser requests the same page the next time, it sends the cookie along with the request information. The Web server reads the cookie and extracts its value. It then, process the Web page according to the information contained in the cookie and renders it on the Web browser.

Figure 9.1 illustrates how cookies are transmitted between browser and application.

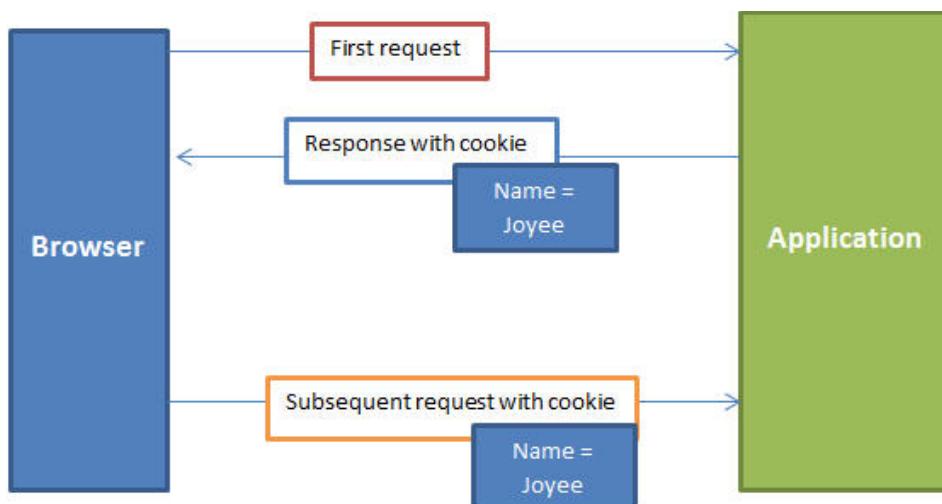


Figure 9.1: Transmission of Cookie between Browser and Application

There are two types of cookies. The first one is the session cookies that are stored in the browser's memory. These cookies are transmitted through the header during every request. This type of cookie is useful for storing information required for only a short time.

The other type of cookie is the persistent cookies that are stored in text files on a user's computer. This type of cookie is useful when you need to store information for a long time.

Code Snippet 1 shows how to create persistent cookies.

Code Snippet 1:

```
Response.Cookies["UserName"].Value = "John";
Response.Cookies["UserName"].Expires = DateTime.Now.AddDays(2);
Response.Cookies["LastVisited"].Value = DateTime.Now.ToString();
Response.Cookies["LastVisited"].Expires = DateTime.Now.AddDays(2);
```

This code creates the `UserName` and `LastVisited` cookies. The `UserName` cookie stores the name of a user and the `LastVisited` cookie stores the date and time when the user last visited the page. The expiry period for both the cookies is set as 2 days.

Note - While creating cookies, if you do not specify the expiry period, by default the cookies will be created as temporary cookies.

Once you have a cookie, you can access the value of the cookie by using the built-in `Request` object. On the other hand to modify a cookie, you need to use the built-in `Response` object.

Code Snippet 2 accesses a cookie that stores a single value.

Code Snippet 2:

```
if (Request.Cookies["UserName"].Value != null)
{
    string Name = Request.Cookies["UserName"].Value;
}
```

This code specifies that if the value of the `UserName` cookie is not null, then, the value is assigned to the `Name` string.

You can also access the values from a cookie that store multiple values. Code Snippet 3 shows how to access a cookie that stores multiple values.

Code Snippet 3:

```
if (Request.Cookies["UserInfo"] != null)
{
    string Name = Request.Cookies["UserInfo"]["UserName"];
```

```
stringVisitedOn = Request.Cookies["UserInfo"]["LastVisited"];
}
```

In this code, the `UserInfo` cookie contains two sub-keys named, `UserName` and `LastVisited`. If the value of the `UserInfo` cookie is not null, the value of the two sub-keys are assigned to the `Name` and `VisitedOn` strings respectively.

9.1.2 TempData

`TempData` is a dictionary that stores temporary data in the form of key-value pairs. You can use `TempData` to store values between requests. You can add values to `TempData` by adding them to `TempData` collection. The data stored in `TempData` is available during the current and subsequent requests. `TempData` is useful in situations when you need to pass data to a view during a page redirect.

Code Snippet 4 shows the content of a controller class named `HomeController`.

Code Snippet 4:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        TempData["tempText"] = "Welcome to MVC";
        return RedirectToAction("Browse");
    }

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

In this code, when you send request for the `Index()` action method, the value, `Welcome to MVC`, is stored in the `TempData` key, named `tempText`. Hence, you are automatically redirected to the `Browse()` action method that will render a view.

Code Snippet 5 shows the content of the `Browse()` action method.

Code Snippet 5:

```
@{var msg = TempData["tempText"] as String;}
@msg
```

The code renders a view that access the data stored for it in `TempData` and then, display it to the user.

9.1.3 Application State

Application state enables storing application-specific information as key-value pairs.

When a user accesses any URL, application state is created for the first time. After that it stores the application-specific information. This information can be shared with all the pages and user sessions of the application. For this, you need to use the `HttpApplicationState` class. This class is accessed by using the `Application` property of the `HttpContext` object.

Whenever any application events occur, the application state is initialized and manipulated. These application events include the following:

- **Application.Start:** Event is raised when an application receives a request for a page for the first time, and when the server or the application is restarted. Event handler of this event contains the code for initializing the application variables.
- **Application.End:** Event is raised when the server or the application is stopped or restarted. Event handler of this event contains the code to clear the resources that the application has already used.
- **Application.Error:** Event is raised when an unhandled error occurs.

The handlers for these three types of events are defined in the `Global.asax` file. You can write the code to manipulate the application state in one of these event handlers.

→ Using Application State

To store application-specific information in application state, you need to create variables and objects. Then, you can add these variables and objects to the application state. These variables and objects are accessible for any components of an ASP.NET MVC application.

You need to write the code for initializing these application variables and objects inside the `Application_Start()` function available in the `Global.asax` file.

Code Snippet 6 shows creating a variable named `TestVariable` and then, stores it in the application state.

Code Snippet 6:

```
HttpContext.Application["TestVariable"] = "Welcome to MVC";
```

This code creates a variable named, `TestVariable`. All the pages and the user sessions of the application can access the value stored in this variable.

When you run the application that includes the newly created variable, any pages of this application can retrieve the value of this variable.

Code Snippet 7 shows how to access the value of TestVariable.

Code Snippet 7:

```
stringval = (string)HttpContext.Application["TestVariable"];
```

This code access the value included in the `TestVariable` and then, stores it in a local variable named `val`.

Once you have created and added a variable or object to an application state, you can also remove it from the application state. For that you can use the `Remove()` method.

Code Snippet 8 shows how to remove an application variable named, `TestVariable`, from the application state.

Code Snippet 8:

```
HttpContext.Application.Remove("TestVariable");
```

This code uses the `Remove()` method that will remove the variable named, `TestVariable` from the application state.

You can also use the `RemoveAll()` method to remove all the available variables present in the application state.

Code Snippet 9 shows using the `RemoveAll()` method.

Code Snippet 9:

```
HttpContext.Application.RemoveAll();
```

This code will remove all the existing variables or objects from the application state.

Note - Once you have added an object to an application state, this object remains in the application state until the application is shut down, the `Global.asax` file is modified, or it is explicitly removed from the application state.

9.1.4 Session State

A session state stores session-specific information for an ASP.NET MVC application. However, the scope of session state is limited to the current browser session. When many users access an application simultaneously, then, each of these users will have a different session state.

Similar to application state, a session state stores application-specific data in key-value pairs. These session-specific information should be maintained between server round trips and requests for pages.

→ Using Session State

An active session is identified and tracked by a unique session ID string containing American Standard Code for Information Interchange (ASCII) characters. The ASP.NET MVC Framework provides the `SessionStateModule` class that enables you to generate and obtain the session ID strings.

Like application state, you can also store objects and variables in a session state. However, a session variable remains active for 20 minutes without any user interaction.

You can use the same methods as application state, to add and access variable or objects from the session state.

Code Snippet 10 shows adding a variable named, `TestVariable` in a session state.

Code Snippet 10:

```
Session["TestVariable"] = "Welcome to MVC Application";
```

This code creates a variable named, `TestVariable` that contains 'Welcome to MVC Application' as its value.

Code Snippet 11 shows the code to access the value of the newly created variable.

Code Snippet 11:

```
stringval = (string) Session["TestVariable"];
```

This code will access the value of `TestVariable` and stores it in the variable named, `val`.

While using session state, you need to consider the following issues:

- ➔ A variable or an object that is added to a session state remains until a user closes the browser window. By default, a variable or object is automatically removed from the session state after 20 minutes if a user does not request a page.
- ➔ A variable or an object added to the session state is related to a particular user.

Similar to application state, a session state can be globally accessed by the current user. A session state can be lost in case of the following situations:

- User closes or restarts the browser.
- User accesses the same Web page through a different browser window.
- The session times out because of inactivity.
- The `Session.Abandon()` method is called within the page code.

To remove an object or a variable that has been added to a session state, you can use the `Remove()` or `RemoveAll()` methods.

While using session states in an application, you need to configure it in the `Web.config` file of the application. The `Web.config` file enables you to define advanced options, such as the timeout and the session state mode.

Code Snippet 12 shows the options that you can configure inside the `<sessionState>` element.

Code Snippet 12:

```
<sessionState
    cookieless="UseCookies" cookieName="Test_SessionID"
    timeout="20"
    mode="InProc" />
```

This code uses the `cookieless`, `timeout`, and `mode` session state attributes.

→ Attributes of Session State

You can use the `cookieless` attribute to specify whether a cookie is to be used or not. Table 9.1 lists the values that you can specify for the `cookieless` attribute.

Value	Description
<code>UseCookies</code>	Allows you to specify that cookies are always used.
<code>UseUri</code>	Allows you to specify that cookies are never used.
<code>UseDeviceProfile</code>	Allows you to specify that the application should check the <code>Browser Capabilities</code> object to identify whether to use cookies.
<code>AutoDetect</code>	Allows you to specify that cookies should be used if the browser supports them.

Table 9.1: Values of Cookieless Attribute

You can use the `timeout` attribute of session state to specify time period in minutes. The application use this specified time to wait for receiving a request. If it does not receive a request from user within this specified time it removes the session.

You can use the `mode` attribute to specify where the values of the session state is to be stored. You can use one of the following values of the `mode` attribute:

- **Custom:** Allows you to specify that a custom data store should be used to store the session-state data.
- **InProc:** Allows you to specify that the data is to be stored in the same process as the application. The `Session.End` event is raised only in this mode.
- **Off:** Allows you to specify that the session state is disabled.
- **SQLServer:** Allows you to specify that the session state is to be stored by using SQL Server database to store the state data.
- **StateServer:** Allows you to specify that the session state is to be stored in a service running on the server or a dedicated server.

9.2 Optimizing Web Application Performance

You can use a Web application for different purposes, such as discussion forums, online shopping, and social networking. You have learned that the performance of an application depends on the fact how quickly they can respond to a user request.

So, to improve the performance of an application, you can reduce the number of interaction between the server and the database. The ASP.NET MVC Framework provides different techniques to improve the performance of an application.

There are two techniques that you can use to improve performance of an application. The first one is caching that you can use to optimize the performance of an application. Caching enables reducing the number of interactions between the server and database. To do this, it stores the data in the cache memory for a specific period of time. The second one is bundling and minification, which allows you to reduce the number of requests and the size of the requests between a client and a server.

9.2.1 Output Caching

Output caching allows you to cache the content returned by an action method. As a result, the same content does not need to be generated each time the action methods invoked.

You need to consider the following factors, before implementing output caching:

- Check whether output caching should be used in the application. If the content of the application changes frequently, output caching may not be useful.
- Check the time duration for which the output has to be cached. This would depend on how frequently data associated with the output changes.
- Check the location where you need to cache the output. You can cache the output on different locations, such as client machines and server.

In an ASP.NET MVC application, you can use output caching to cache the pages of the application for a specific time period. When a user request arrives for a same page within that time period, the application renders the cached page to the user instead of re-rendering the page.

Figure 9.2 shows how ASP.NET MVC implements output caching.

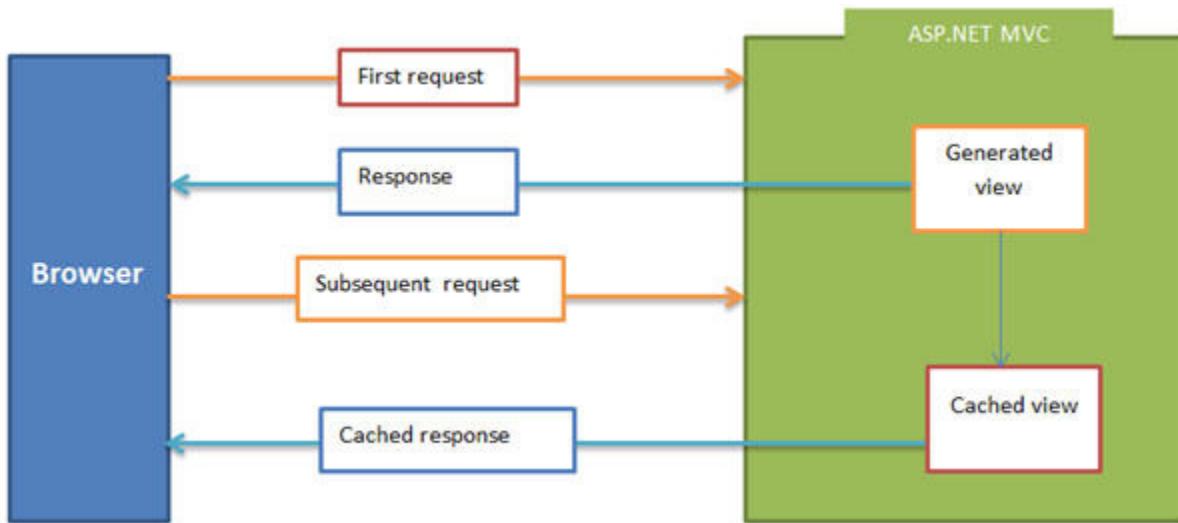


Figure 9.2: Output Caching in ASP.NET MVC

In an application, you can use caching at an action method. You can use the `output cache` attribute to an action method or the whole controller that needs to be cached. Thus, you can cache the result returned by the action method. This results getting same content every time when a new user invokes the same action method.

Code Snippet 13 shows how to add an output cache attribute to an action method.

Code Snippet 13:

```

public class HomeController : Controller
{
    [OutputCache]
    public ActionResult Index()
    {
        //Code to perform some operations
    }
}
  
```

In this code, the `[OutputCache]` attribute is added to the `Index()` action method of the Home controller.

You can also use caching to cache the output for a specified duration. Caching the output for a small period of time is known as micro caching that you can use when the traffic on an application is high. This type of caching enables you to reduce the number of queries to access the database.

Code Snippet 14 shows specifying the duration and location of a cache attribute.

Code Snippet 14:

```
public class HomeController : Controller
{
    [OutputCache(Duration=5)]
    public ActionResult Index()
    {
        ViewBag.Message = "This page is cashed for " + DateTime.Now;
        return View();
    }
}
```

In this code, the output is cached for 5 seconds. This will refresh the page in every five seconds.

You can also define the location where you want the data to be cached. By convention, the data is cached in three locations when you use the `[OutputCache]` attribute. These locations can be the server, a proxy server, and the Web browser.

However, you can specify a location of the output to be cached. For this, you need to modify the `Location` property of the `[OutputCache]` attribute. You can use any one of the following values for the `Location` property:

- **Any**: Specifies that the output cache should be stored on the browser, a proxy server, or the server where the request was processed.
- **Client**: Specifies that the output cache should be stored on the browser where the request originated.
- **Downstream**: Specifies that the output cache should be stored in any HTTP cache-capable device other than the original server.
- **Server**: Specifies that the output cache should be stored on the server where the request was processed.
- **None**: Specifies that the output cache is disabled for the requested page.

Consider a scenario, where you are caching personalized data for each of the users accessing your application. In such scenario, you need to cache the data in the browser cache to improve performance, instead of caching the data on the server.

Code Snippet 15 shows how to set the Location property.

Code Snippet 15:

```
public class ProductController : Controller
{
    [OutputCache(Duration=36000,
    Location=System.Web.UI.OutputCacheLocation.Client)]
    public string GetProductDetails()
    {
        //Code to display product details
    }
}
```

In this code, the `[OutputCache]` attribute allows caching the output of the `GetProductDetails()` action method. Then, the `Location` property is set to the `System.Web.UI.OutputCacheLocation.Client` value. This allows caching the data on the browser, so that when different users invoke the `GetProductDetails()` action method, each user gets its own product details.

9.2.2 Data Caching

In recent times, most of the Web applications contain dynamic data. These data needs to be retrieved from a database. So executing a database query for different users every time can reduce the performance of the application. To overcome such problems, you can use the data caching technique. To perform data caching in an ASP.NET MVC application, you can use the `MemoryCache` class.

Code Snippet 16 shows how to use the `MemoryCache` class to cache data.

Code Snippet 16:

```
Customer dtUser = System.Runtime.Caching.MemoryCache.Default.
AddOrGetExisting("UserData", getUser(),
System.DateTime.Now.AddHours(1));
```

In this code, the `AddOrGetExisting()` method enables the application to refresh and access data from the cache. This method accesses the data from the cache if it contains the relevant data. If there is no relevant data in the cache, then, the `AddOrGetExisting()` method allows adding the data to the cache by invoking the `getUser()` method.

The first parameter of the `AddOrGetExisting()` method specifies the unique identifier of the object that needs to be stored in the memory cache. The second parameter specifies the object that needs to be stored in the cache, and the third parameter specifies the time when the cache should expire.

You can use a HTTP caching technique in the browser and proxy cache. Storing data in the browser cache allows you to reduce the process of downloading content from the server repeatedly. Typically, a Web browser frequently checks for content updates and then, downloads the content from the server to fulfill user requests. Otherwise the Web browser renders the content from the local cache.

9.3 Bundling and Minification

The ASP.NET MVC Framework provides two techniques known as Bundling and Minification. These techniques allow you to improve the load time of a Web page in an application. This is done by reducing the number of user requests to the server and the size of the requested resources.

9.3.1 Using Bundling

Bundling is a technique that allows you to reduce the user requests in your application by combining several individual scripts into a single request. Thus, it reduces the number of requests between the client and a server.

When there are multiple files that need to be downloaded, at times it became very time consuming. As a solution, you can use the bundling technique. This technique allows you to combine multiple files and then, it can be downloaded as a single entity.

To use the bundling technique in an application, you need to define the files that you want to combine together. You can achieve this using the `BundleConfig` class in the `App_Start` folder of the application. You can create a bundle using the `RegisterBundles()` method of the `BundleConfig` class.

Code Snippet 17 shows using the `RegisterBundles()` method of the `BundleConfig` class.

Code Snippet 17:

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
        "~/Scripts/jquery.unobtrusive*",
        "~/Scripts/jquery.validate*"));

    bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/Site.css",
        "~/Content/styles.css"));
}
```

In this code, the `RegisterBundles()` method creates, registers, and configures bundles. A new instance of the `ScriptBundle` class is created to create a bundle of scripts. The path, `~/bundles/jqueryval`, has been assigned to the bundle. Then, the `Include()` method on the bundle object enables you to specify the files that should be included in the bundle. Finally, the `*` symbol is used to indicate that the bundle should include all the script files in the `Scripts` folder whose name begins with `jquery.unobtrusive` or `jquery.validate`.

You can include these bundled files in a view. For that, you need to use the `@Styles.Render` and `@Scripts.Render` methods in your view. After that, you need to specify a path that you have configured in the bundle configuration as a parameter to these methods.

Code Snippet 18 shows using the `@Styles.Render` and `@Scripts.Render` methods in a view.

Code Snippet 18:

```
@Scripts.Render("~/bundles/jqueryval")  
@Styles.Render("~/Content/css")
```

9.3.2 Using Minification

Minification allows you to reduce the size of a file by removing unnecessary white spaces and comments, and shortening the variable names. You can use such code while using interpreted languages, such as JavaScript.

By using minification, you can reduce the size of a file and thus reduce the time required to access it from the server and load it into the browser.

9.4 Check Your Progress

1. Which of the following options will you use to access the value of a cookie?

(A)	A Request object
(B)	A Response object
(C)	A TempData object
(D)	The BundleConfig class
(E)	The MemoryCache class

(A)	A	(C)	D
(B)	E	(D)	B

2. Which of the following options in an application allows containing information, such as the IP address, browser type, operating system, and Web pages last visited?

(A)	TempData
(B)	Bundling
(C)	Cookies
(D)	MemoryCache
(E)	BundleConfig

(A)	B	(C)	D
(B)	C	(D)	A

3. Which of the following options will you use to generate and obtain the session ID strings?

(A)	Session.Abandon() method
(B)	HttpApplicationState class
(C)	HttpContext object
(D)	SessionStateModule class
(E)	Application_Start() function

(A)	B	(C)	A
-----	---	-----	---

(B)	A	(D)	D
-----	---	-----	---

4. Which of the following code snippets correctly access the value included in the TestVariable and then, stores it in a local variable named val?

(A)	string val = (string)HttpAppState. Application["TestVariable"];
(B)	string val = (string)HttpRequest. Application["TestVariable"];
(C)	string val = (string)HttpContext.Application[@ TestVariable];
(D)	string val = (string)HttpSession. Application["TestVariable"];
(E)	string val = (string)HttpContext. Application["TestVariable"];

(A)	B	(C)	D
(B)	C	(D)	E

5. Which of the following values of the mode attribute allow you to specify that the data is to be stored in the same process as the application?

(A)	SQLServer
(B)	Custom
(C)	InProc
(D)	Any
(E)	StateServer

(A)	B	(C)	A
(B)	C	(D)	D

9.4.1 Answers

(1)	A
(2)	B
(3)	D
(4)	D
(5)	B



Summary

- ASP.NET MVC Framework allows you to use layouts to simplify the process of maintaining consistent look and feel in an application.
- The _Layout.cshtml file represents the layout that you can apply to the views of an application.
- You can specify a layout for a view by using the Layout property.
- A nested layout refers to a layout that is derived from a parent layout and is responsible for defining the structure of a page.
- Styles are used to define a set of formatting options, which can be reused to format different HTML helpers on a single view or on multiple views.
- The Site.css file allows you to define a particular style that needs to be applied in the application.
- Adaptive rendering enables automatic scaling down of a page and redrawing the page according to a device screen.

Session - 10

Authentication and Authorization

Welcome to the Session, **Authentication and Authorization**.

The ASP.NET MVC Framework provides various mechanisms that you can use to restrict user access to specific content in your application.

ASP.NET MVC 5 uses ASP.NET Identity that you can use to add login features to your application. This is a new membership system for ASP.NET MVC applications.

While developing an ASP.NET MVC application, you can also use the authorization mechanism to verifying whether an authenticated user has the privilege to access a requested resource.

An ASP.NET MVC application also enables you to authorize some specific users and roles.

In this Session, you will learn to:

- ➔ Define and describe authentication
- ➔ Explain and describe how to implement authentication
- ➔ Define and describe ASP.NET Identity
- ➔ Define and describe the process of authorization

10.1 Authentication

Consider a scenario where, you are developing an ASP.NET MVC application. This application allows users to view and buy products. On this application, though available products can be viewed by any users, you want that only registered user can buy a product. So, a user who wants to buy products first requires registering to the application and then, can buy any product. For that, you need to verify a user so that they are able to buy a product. In such situation, you can identify a user by using the authentication mechanism.

Authentication is the process of validating the identity of a user before granting access to a restricted resource in an application. When the application receives a request from users, it tries to authenticate the user. Typically, authentication allows identifying an individual based on the user name and password provided by the user.

10.1.1 Authentication Modes

The ASP.NET MVC Framework enables you to use three types of authentication modes, such as Forms authentication, Windows authentication, and OpenID/OAuth.

→ Forms Authentication

In an ASP.NET MVC application, when you use the Forms authentication, the application itself is responsible for collecting user's credentials and then, authenticates the user. Here, the application can authenticate the users by providing a login form on the Web page. In this page, a user can specify a user name and password to login. When a user successfully logs into the application and is authenticated to view a Web page, it generates a cookie that is served as an authentication token to the user. This cookie is then, used by the browser for the future requests made by the user, and thus allows the application to validate the requests.

When you create an ASP.NET MVC application in Visual Studio 2013, the application does not automatically uses any type of authentication. So, to authenticate users by using Forms authentication, you need to manually configure the `<authentication>` element under the `<system.web>` element present in the `Web.config` file.

Code Snippet 1 shows the default `<authentication>` element in the `Web.config` file.

Code Snippet 1:

```
<system.web>
  <authentication mode="None" />
</system.web>
```

In this code, the `<authentication>` element contains the `mode` attribute that allows you to specify what type of authentication is to be used by the application.

Now, to use Forms authentication mode, you need to change the `mode` attribute of the `<authentication>` element to `Forms`. Code Snippet 2 shows specifying the authentication mode as `Forms`.

Code Snippet 2:

```
<system.web>
    <authentication mode="Forms" />
</system.web>
```

In this code, the `mode` attribute of the `<authentication>` element is set to `Forms`, which indicates that application should use the Forms authentication.

→ **Windows Authentication**

This type of authentication is best suited to an intranet environment where a set of known users are already logged on to a network. Consider a scenario, where you are developing an application that is to be used by the employees of an organization. Here, you can use the Windows Authentication mode to authenticate the employees. This is because the existing employees already have their accounts in the database of the Web server. So, the application that uses the Windows Authentication rejects an employee's request that is not an authenticated employee or has an invalid user name or password.

If you want to use the Windows Authentication mode in your application, it is the firewall that takes care of the authentication process. In this mode, the application uses the users credential authenticated by the company's firewall for security checks.

To configure your application to use the Windows Authentication mode, you need to change the `mode` attribute of the `<authentication>` element to `Windows`.

Code Snippet 3 shows specifying the authentication mode as Windows.

Code Snippet 3:

```
<system.web>
    <authentication mode="Windows" />
</system.web>
```

In this code, the `mode` attribute of the `<authentication>` element is set to `Windows`, which indicates that application should use the Windows Authentication mode.

- **OpenID/OAuth**

The OAuth and OpenID are authorization protocols that enable a user to logon to an application by using the credentials for third party authentication providers, such as Google, Twitter, Facebook, and Microsoft.

When you create an ASP.NET MVC Web application using Visual Studio 2013, the third party authentication providers need to be configured. You can do this in the `Startup.Auth.cs` file present in the `App_Start` folder in the application directory. By default, the `Startup.Auth.cs` file contains the commented code that enables logging in with third party login providers.

Code Snippet 4 shows the content of the `Startup.Auth.cs` file.

Code Snippet 4:

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;
namespace SecureApp
{
    public partial class Startup
    {
        // For more information on configuring authentication, please visit
        // http://go.microsoft.com/fwlink/?LinkId=301864
        public void ConfigureAuth(IAppBuilder app)
        {
            // Enable the application to use a cookie to store information for the
            // signed in user
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
                LoginPath = new PathString("/Account/Login")
            });
            // Use a cookie to temporarily store information about a user logging
            // in with a third party login provider
            app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);
            // Uncomment the following lines to enable logging in with third party
            // login providers
            //app.UseMicrosoftAccountAuthentication(
            //    clientId: "",
            //    clientSecret: "");
            //app.UseTwitterAuthentication(
            //    consumerKey: "",
```

```
// consumerSecret: "");

//app.UseFacebookAuthentication(
//    appId: "",
//    appSecret: "");

//app.UseGoogleAuthentication();
}

}
```

This code shows content of the `Startup.Auth.cs` file where the third party login provider's code is commented.

As you have seen that the third party login provider's code is commented, because before you can use the services provided by an external login providers, you have to register the application with them. To use the services of Google, you are not required to register with the ASP.NET MVC application if you are already registered with Google. To use the services of Google, you only need to uncomment the call to the `OAuthWebSecurity.RegisterGoogleClient()` method. When you run the default application created by Visual Studio 2013 and click the Log in link on the home page, the application will provide the option to log in using Google credentials. Figure 10.1 shows the Login page.

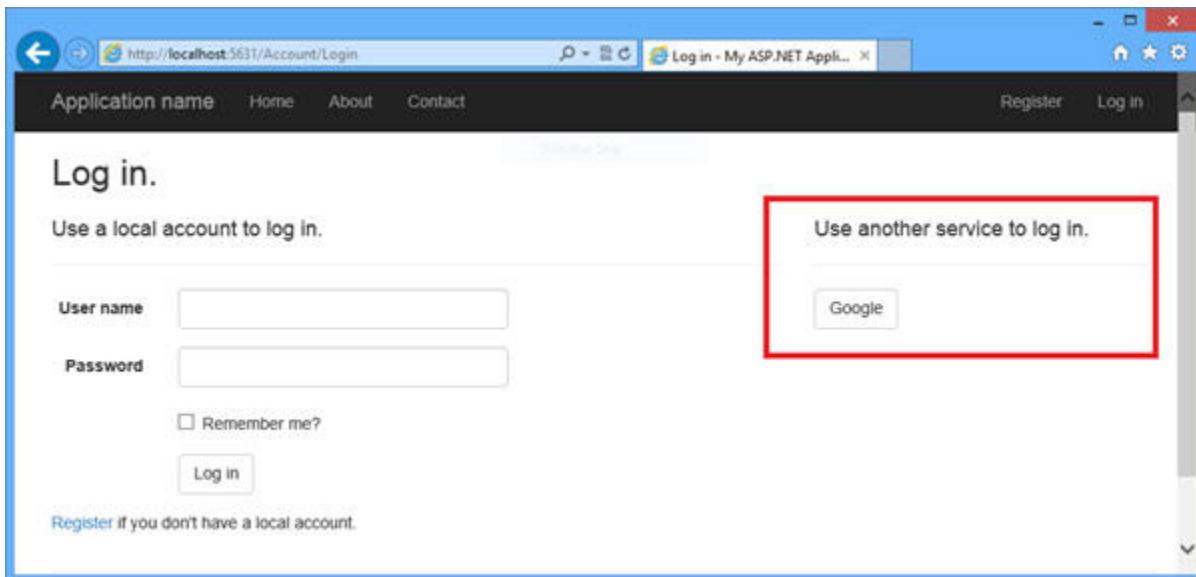


Figure 10.1: Login Page

10.1.2 Enforcing Authentication Using the Authorize Attribute

When a proper security mechanism is implemented on an application, a user must be enforced to login to the application to access some specific resources. This type of mechanism is implemented in an application by using the `Authorize` action filter. You can use these attributes to add behavior that can be executed either before an action method is called or after an action method is executed.

In an ASP.NET MVC application, you can use the `Authorize` attribute to secure an action method, a controller, or the entire application.

→ Securing a Controller Action

Consider a scenario, where you are developing an online shopping store application. In the application, you want to display the list of available items in the home page from where a user can select an item and purchase it. Now, for security reason, you want that all users can see the list of item on the home page, but only the authenticated user can purchase an item using this application. In such situation, you need to create a `BuyItem()` action method in a controller class, named `Product`.

Now, while creating the `BuyItem()` action method, you should ensure that this method is only accessible for the authenticated users. For this, you need to add the `Authorize` attribute on the `BuyItem()` action method.

Code Snippet 5 shows adding the `Authorize` attribute the `BuyItem()` action method.

Code Snippet 5:

```
[Authorize]
public ActionResult BuyItem()
{
    return View();
}
```

In this code, the `Authorize` attribute is added on the `BuyItem()` action method.

→ Securing a Controller

You can also use the `Authorize` attribute to a controller to secure the entire controller. For this, you need to add the `Authorize` attribute on the controller.

Code Snippet 6 shows adding the `Authorize` attribute to a controller.

Code Snippet 6:

```
[Authorize]
public class ProductController : Controller
{
```

```
...
}
```

The code will restricts all the action methods defined in the `ProductController` controller class.

Sometimes, you might not want to secure certain action methods in a controller. In such case, you can use the `AllowAnonymous` attribute on those action methods.

Code Snippet 7 shows using the `AllowAnonymous` attribute.

Code Snippet 7:

```
[AllowAnonymous]
public ActionResult Index()
{
    return View();
}
```

This code uses the `AllowAnonymous` attribute on the `Index()` action. As a result, access to this method does not require authentication.

At times, you may want that all the users to be authenticated for an entire application. For this, you need to add the `Authorize` attribute as a global filter. You can use the `Authorize` attribute as a global filter, by adding it to the global filters collection in the `RegisterGlobalFilters()` method in the `FilterConfig.cs` file.

Code Snippet 8 uses the `RegisterGlobalFilters()` method in the `FilterConfig.cs` file.

Code Snippet 8:

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new System.Web.Mvc.AuthorizeAttribute());
    filters.Add(new HandleErrorAttribute());
}
```

This code applies the `Authorize` attribute to all action methods in the application. However, you can use the `AllowAnonymous` attribute to allow access to specific controllers or action methods.

10.1.3 ASP.NET Identity

Prior to ASP.NET MVC 5, the MVC Framework used the Membership API for implementing authentication and authorization. ASP.NET MVC 5 uses ASP.NET Identity, which is a new membership system for ASP.NET MVC applications. You can use ASP.NET Identity to add login features to your application.

ASP.NET Identity uses the Code First approach to persist user information in a database.

When you create an ASP.NET MVC 5 application in Visual Studio 2013, the `Identity` and `Account` management classes are automatically added to the project. The `IdentityModel.cs` file created by Visual Studio 2013 manages the identity of an application.

Code Snippet 9 shows the `IdentityModel.cs` file.

Code Snippet 9:

```
namespace WebApplication1.Models
{
    public class ApplicationUser : IdentityUser
    {
    }

    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext()
            : base("DefaultConnection")
        {
        }
    }
}
```

In this code, there are two classes. The `ApplicationUser` class that extends from the `IdentityUser` class is responsible for managing user accounts in the application. The second class is `ApplicationDbContext` that inherits from `IdentityDbContext`. The `ApplicationDbContext` class allows the application to interact with the database where account data of users are stored.

You will now learn how ASP.NET Identity is used in a default ASP.NET MVC application created using Visual Studio 2013. When you execute the default application, the Home page appears.

Figure 10.2 shows the Home page.

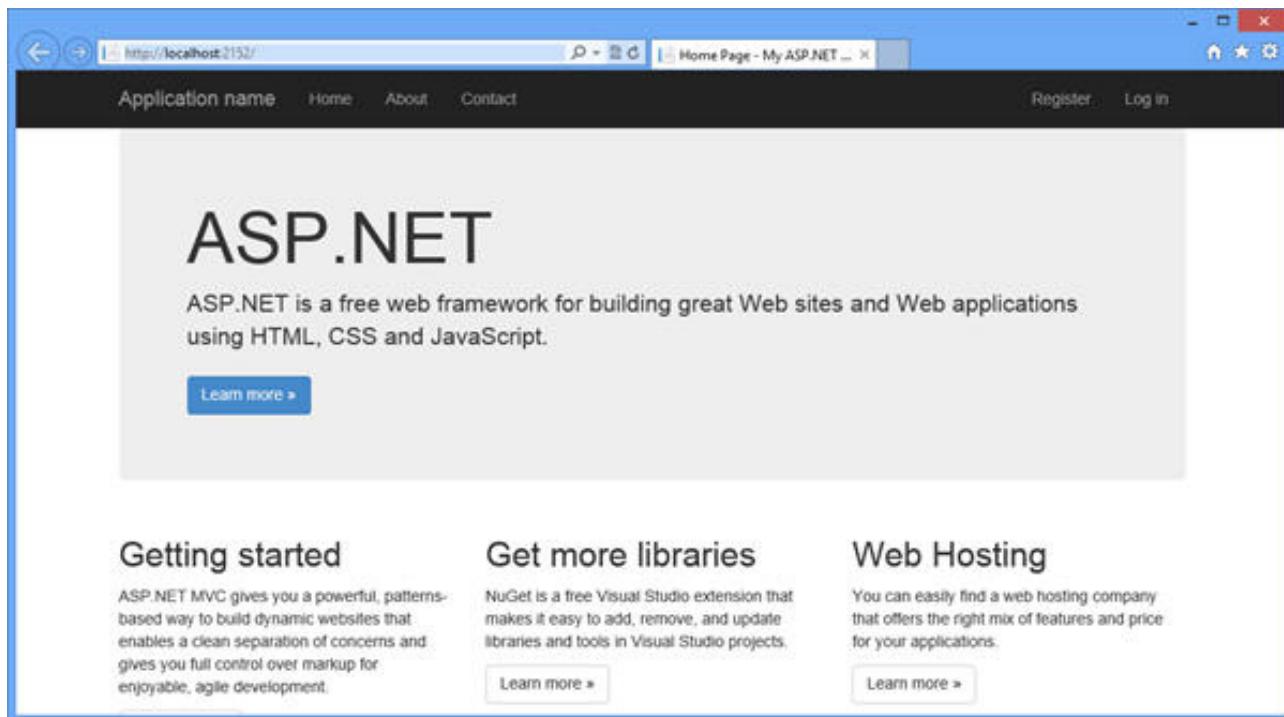


Figure 10.2: Home Page

1. Click the **Register** link. The **Register** page appears.
2. Add the registration details. Figure 10.3 shows specifying registration details.

A screenshot of a web browser displaying the "Register" page. The URL in the address bar is "http://localhost:2152/Account/Register". The page has a dark header with "Application name" and navigation links for "Home", "About", and "Contact". On the right, there are "Register" and "Log in" links. The main content area has a heading "Register." and a sub-instruction "Create a new account." Below this are three input fields: "User name" (containing "Sam"), "Password" (containing "*****"), and "Confirm password" (containing "*****"). A "Register" button is at the bottom. At the very bottom of the page, a small copyright notice reads "© 2014 - My ASP.NET Application".

Figure 10.3: Specifying Registration Details

- Click the **Register** button. The application logs you on and the home page is displayed with a welcome message based on the specified **user name** and a **Log off** option. Figure 10.4 shows the welcome message.

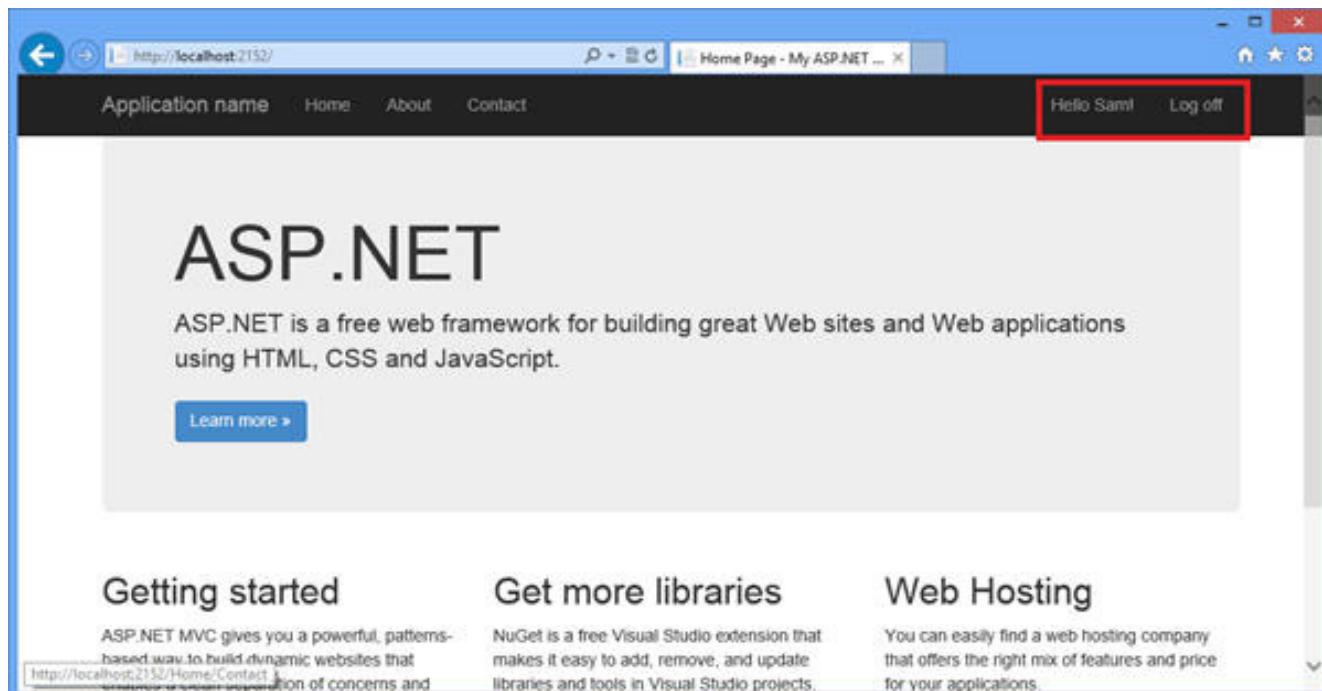


Figure 10.4: Welcome Message

Internally, the application uses ASP.NET Identity to create database tables for the application to hold account data. You can view the generated tables by performing the following tasks:

- Select **View→Server Explorer**. The **Server Explorer** window appears.
- Select **DefaultConnection (WebApplication1)** under the **Data Connections** node and click the **Add SQL Server** icon. Figure 10.5 shows the **Server Explorer** window.

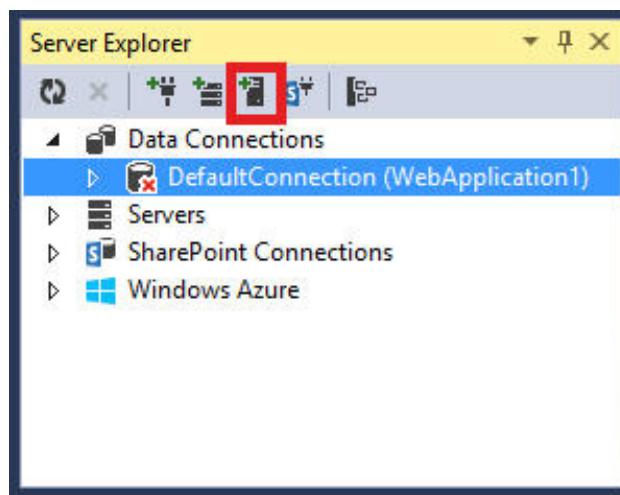


Figure 10.5: Server Explorer Window

3. In the **Connect to Server** dialog box that appears, type `(LocalDb)\v11.0` in the Server name text field. Figure 10.6 shows the **Connect to Server** dialog box.



Figure 10.6: Connect to Server Dialog Box

4. Click **Connect**.
5. Expand the **Default Connection (WebApplication1)→Tables** node to view the tables that are automatically generated by Visual Studio 2013 based on ASP.NET Identity.
6. Figure 10.7 shows the tables that are automatically generated by Visual Studio 2013.

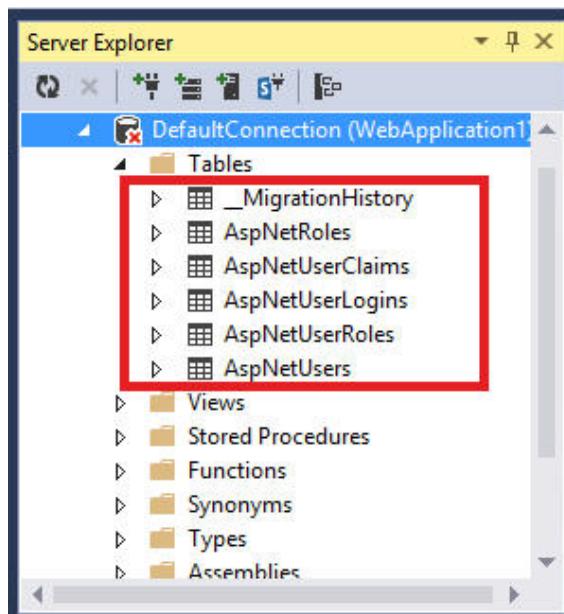


Figure 10.7: Tables Created by Visual Studio 2013

10.2 Authorization

Authorization is the process of verifying whether an authenticated user has the privilege to access a requested resource. After creating users that can access your application by using the membership management service, you need to specify authorization rules for the users.

You need to grant different permissions to different users to provide accessibility to the pages on your application. For example, on an online shopping store application, the users are able to view the list of available products. However, they cannot make any changes to the products that are displayed on a page of the application. However, a user with the administrator role in the application can modify the product list, such as add or remove any product from the list. This requires giving different levels of permission to different users of the application.

10.2.1 Users and Roles

In order to maintain a different set of settings for each user is a complex task. So, you should assemble the users in a group user known as roles and then, define the permissions on these roles.

In an ASP.NET MVC application, you can authorize some specific users and roles by using the `Authorize` attribute.

Code Snippet 10 uses the `Authorize` attribute to authorize specific users.

Code Snippet 10:

```
[Authorize(Users="Mathews, Jones")]
public class ManagerProduct : Controller
{
    // Code to perform some operation
}
```

In this code, only the users whose name is `Mathews` and `Jones` are the authorized users to access the `ManagerProduct` controller.

In an ASP.NET MVC application, you can also assign roles to different users. Thereafter, you can use the authorization at the role level.

Code Snippet 11 shows specifying role to users.

Code Snippet 11:

```
[Authorize(Roles="Administrator")]
public class ManagerProduct : Controller
{
    // Code to perform some operation
}
```

This code allows accessing the ManagerProduct controller to users with the Administrator role.

You can also use a combination of roles and users. Code Snippet 12 shows how to use a combination of roles and users.

Code Snippet 12:

```
[Authorize(Roles="Manager", Users="Steve, Mathews")]
public class ManagerProduct : Controller
{
    // Code to perform some operation
}
```

This code allows the users whose name is Steve and Mathews, and users with the Manager role to access the ManageStore controller.

10.2.2 Role Providers

You need to create roles so that you can assign specific rights to specific users. The ASP.NET MVC Framework provides different role providers that you can use to implement roles. Some of them are as follows:

- **ActiveDirectoryRoleProvider:** This provider class enables you to manage role information for an application in the Active Directory.
- **SqlRoleProvider:** This provider class enables you to manage role information for an application in SQL database.
- **SimpleRoleProvider:** This role provider works with different versions of SQL server.
- **UniversalProviders:** This provider works with any database that Entity Framework supports.

You can use the `Provider` property provided by the `Roles` class to access the current role provider.

Code Snippet 13 shows uses the `Provider` property.

Code Snippet 13:

```
var roles = (TestRoleProvider)Roles.Provider;
```

In this code, the `Provider` property allows to access the role provider, named `TestRoleProvider`. This piece of code will only work, when the `System.Web.Security` namespace is included.

10.2.3 User Accounts and Roles

When you create an MVC application using Visual Studio 2013, you need to create the membership database with a connection string that you need to specify in the `Web.config` file.

You can add sample data in this database with the required roles by seeding the membership database. To do this, you need to use the `Seed()` method.

Code Snippet 14 shows using the `Seed()` method.

Code Snippet 14:

```
var roles = (SimpleRoleProvider)Roles.Provider;
if (!roles.RoleExists("Administrator"))
{
    roles.CreateRole("Administrator");
}
```

In this code, you first retrieve a reference of the simple role provider. Then, you use this reference to check whether the `Administrator` exists in the database. If it does not exist, the `CreateRole()` method creates the role named, `Administrator`.

Once you have created an `Administrator` role in the database, you can add users to that role.

Code Snippet 15 shows assigning a user with the `Administrator` role.

Code Snippet 15:

```
if (!roles.GetRolesForUser("Mark_Jones").Contains("administrator"))
{
    roles.AddUsersToRoles(new[] { "Mark_Jones" }, new[] { "administrator" });
}
```

In this code, if the user with the name `Mark_Jones` is already assigned with the `Administrator` role, then the `AddUsersToRoles()` method adds the `Administrator` role, to the user with the name, `Mark_Jones`.

Once you have created a role and users in the database, you can then, use the `Authorize` attribute on the action method that needs to be restricted.

Code Snippet 16 shows using the `Authorize` attribute so that the action method can be accessed by users with the `Administrator` role.

Code Snippet 16:

```
[Authorize(Roles = "administrator")]
public ActionResult ManageProduct()
{
    return View();
}
```

This code will display a login page when a user tries to access the `ManageProduct` page. When the user

provides the valid credentials for a user with the administrator role then, the ManageProduct page is displayed.

You can also display only those links that a user has rights to access. For example, the ManageProduct link should be visible only to the users with administrator role. You can hide the ManageProduct link from other users using the ActionLink() method.

Code Snippet 17 shows using the ActionLink() method.

Code Snippet 17:

```
@if(User.IsInRole("administrator"))  
{  
@Html.ActionLink("ManageProduct", "ManageProduct")  
}
```

This code displays the ManageProduct link only when a user logs in as an administrator.

10.3 Check Your Progress

1. Which of the following options allows your application itself for collecting user credentials and then, authenticates the user?

(A)	Windows Authentication		
(B)	Forms Authentication		
(C)	ASP.NET Identity		
(D)	OpenID/OAuth		
(E)	Authorization		

(A)	A	(C)	D
(B)	E	(D)	B

2. Which of the following options will you use to specify what type of authentication is to be used by the application?

(A)	mode attribute		
(B)	forms property		
(C)	windows property		
(D)	Startup.Auth.cs file		
(E)	Authorize attribute		

(A)	B	(C)	E
(B)	C	(D)	A

3. Which of the following options will you use to secure certain action methods in a controller?

(A)	OAuthWebSecurity() method		
(B)	RegisterGlobalFilters() method		
(C)	Authorize attribute		
(D)	AllowAnonymous attribute		
(E)	mode attribute		

(A)	D	(C)	A
(B)	C	(D)	E

4. Which of the following role provider works with any database that Entity Framework supports?

(A)	ActiveDirectoryRoleProvider		
(B)	SqlRoleProvider		
(C)	SimpleRoleProvider		
(D)	UniversalProviders		
(E)	EntityProviders		

(A)	B	(C)	D
(B)	C	(D)	E

5. Which of the following methods will you use to add sample data in a database with the required roles?

(A)	CreateRole()		
(B)	Seed()		
(C)	AddUserstoRoles()		
(D)	RegisterGlobalFilters()		
(E)	ActionLink()		

(A)	B	(C)	A
(B)	C	(D)	D

10.3.1 Answers

(1)	D
(2)	D
(3)	A
(4)	C
(5)	A

Summary

- Authentication is the process of validating the identity of a user before granting access to a restricted resource in an application.
- In an ASP.NET MVC application, you can use the Authorize attribute to secure an action method, a controller, or the entire application.
- ASP.NET Identity uses the Code First approach to persist user information in a database.
- Authorization is the process of verifying whether an authenticated user has the privilege to access a requested resource.
- In an ASP.NET MVC application, you can authorize some specific users and roles by using the Authorize attribute.
- The ASP.NET MVC Framework provides different role providers that you can use to implement roles.
- When you create an MVC application using Visual Studio 2013, you need to create the membership database with a connection string.

ASK to LEARN

Questions
in your
mind?



are here to HELP

Post your queries in **ASK to LEARN** @

Session - 11

Security

Welcome to the Session, **Security**.

While developing an ASP.NET MVC application, first you should analyze the possible attacks and unauthorized access associated with your application. Then, you need to implement the appropriate security measures to protect the application from different types of attacks and unauthorized access.

The ASP.NET MVC Framework supports various techniques, such as encryption and decryption to secure data in storage and in transit. Encryption is the process of transforming data into a secured unreadable format. Transformation of this encrypted data into the original format is known as decryption.

In addition, the ASP.NET MVC Framework also supports several techniques, such as salting, hashing, and digital signature that you can use to ensure security of data in your application.

In this Session, you will learn to:

- ➔ Define and describe how to secure applications
- ➔ Define and describe different types of malicious attacks
- ➔ Explain how to protect a Web application against malicious attacks
- ➔ Explain the process of securing application data

11.1 Malicious Attacks and their Preventions

Web applications are deployed so that it can be accessed by different types of users. Some of these users might plan to carry out malicious attacks against these applications. The motive behind these malicious attacks varies, such as a novice user may carry out a malicious attack out of curiosity, whereas an experienced user may carry out a malicious attack in order to access any sensitive data from the application.

Irrespective of the motive behind malicious attacks, while developing an application, you should ensure that a proper security mechanism is used to protect it from such attacks. Some of the common malicious attacks are as follows:

- Cross-site Scripting (XSS)
- Cross-site Request Forgery (CSRF)
- SQL Injection

You need to protect your application from these attacks to ensure that the malicious users cannot execute the code to harm the data, during the execution of the application.

11.1.1 Cross-site Scripting

Consider a scenario where a Web page of your application allows users to submit their e-mail IDs to receive weekly news updates and stores the e-mail IDs in a database table. However, instead of entering an e-mail ID, a user may enter JavaScript code to open an alert box. Further, the user may submit this page, which consequently stores the JavaScript code in a database table. Later, when the administrator of the application attempts to view the e-mail IDs, the JavaScript code is retrieved from the database and is rendered on the Web page. This causes the JavaScript code to execute on the browser displaying an alert box.

These types of attack are referred as XSS. An XSS attack can be of the following two types:

- **Persistent Attack:** In this type of attack, the harmful code is stored in the database.
- **Non-persistent Attack:** In this type of attack, malicious code is not stored in the database; instead this code is rendered on the browser itself. To perform such attack, the attacker enables a user to click a link containing malicious scripts sent through e-mail or in a chat message. When the user clicks the link, the injected code travels to the application, carries out the intended attack, and the response of the attack is reflected on the user's browser.

To prevent XSS attacks, you should ensure that all HTML code that an application accepts from a user is encoded. HTML encoding is a process that converts potentially unsafe characters to their HTML-encoded equivalent. As a result, when you encode HTML inputs submitted to your application, the HTML engine does not interpret these characters as parts of the HTML markup and renders them as strings.

Apart from this, you also require ensuring that all HTML outputs of the application are encoded to prevent XSS attacks. Consider a scenario, where a user has gained access to the database of your application and inserted a script in a table.

Code Snippet 1 shows the script inserted in a table.

Code Snippet 1:

```
<script>alert('Visit the gethacked website')</script>
```

This code displays an alert box on the browser when the application accesses the script and sends it to the browser as response.

In such scenario, you need to encode the HTML output to prevent such attacks. Code Snippet 2 shows encoding HTML output that converts the preceding markup.

Code Snippet 2:

```
&lt;script&gt;alert(&#39;Visit the gethacked website&#39;)&lt;/script&gt;
```

This code will be displayed by the browser instead of executing the JavaScript code.

→ **Built-in Support to Prevent XSS Attack**

The ASP.NET MVC Framework provides a request validation feature that examines an HTTP request to check and prevent potentially malicious content. Consider a scenario, where if a user attempts to submit HTML or JavaScript code to carry out an XSS attack through the body, header, query string, or cookies of the request, the request validation feature will throw an `HttpRequestValidationException` exception.

Figure 11.1 shows an error message that the browser displays, once a form is submitted that contains HTML or JavaScript code to carry out an XSS attack.

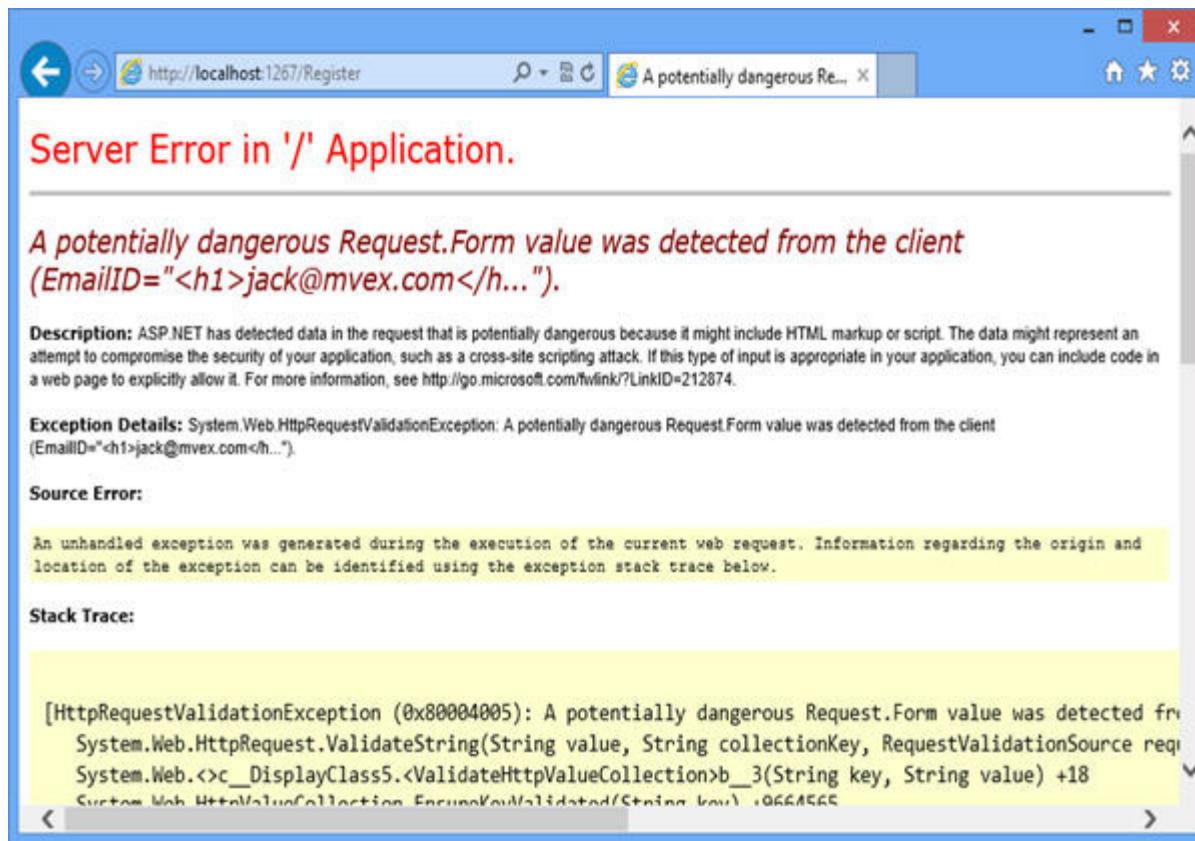


Figure 11.1: Error Message Displayed for a Form Containing HTML Markups

Another built-in feature that you can use in an ASP.NET MVC application to prevent XSS attack is the Razor HTML encoding of the Razor view engine. When you use the @ symbol in a view to refer any data, the Razor HTML encoding feature automatically encodes the data and makes it safe to display as HTML on the browser.

Consider a scenario where you are administering an online shopping store application. In this application, a user named Mark has a membership account. Therefore, he is capable of buying and selling products using the application. Mark is currently logged on to the application and performing transactions. In this transaction, the application uses the browser cookies to authenticate Mark, each time he performs a new transaction on the application. Now, a user acquainted with the working of the application creates an image as a clickable button that generates a request.

Code Snippet 3 shows the generated request when a user clicks the button.

Code Snippet 3:

<http://sales.example.com/transaction/sale?account=mark&productid=P412>

This code is generated when Mark clicks the image that the attacker sends through an e-mail. If Mark is currently logged in on the online shopping application, then the application performs the transaction initiated by the request. Thus, a product is sold without Mark's knowledge.

This is an example of a CSRF attack that tricks an authenticated user to perform some action unknowingly on an application. There are two approaches that you can use to block CSRF attacks, such as domain referrer and user-generated token.

→ Domain Referrer

This approach checks whether an incoming request has a referrer header for your domain. This enables you to ensure that the request has only been initiated from your application. Thus, it prevents any requests coming from unknown sources.

While developing an application, you can use the domain referrer approach by checking whether a user that posts data through a form is only from your application. Consider a scenario where the online shopping store application allows a new user to register. In the application, the `RegisterUser()` action method stores registration data submitted by a user to a database. Now, you want that the registration data is only submitted from your application and not from some other application. To achieve this, you can use the `Request.Url.Host` property in the action method to retrieve the host name of your application. Thereafter, you can retrieve the host name of the referrer by using the `Request.UrlReferrer.Host` property.

Code Snippet 4 shows comparing both the host names and throws an exception if they do not match.

Code Snippet 4:

```
string referrerHostName = Request.UrlReferrer.Host;
string appHostName = Request.Url.Host;
if (appHostName != referrerHostName)
{
    throw new UnauthorizedAccessException();
}
```

In this code, the host name of the application and the referrer is retrieved and compared. If both the host names do not match, then the Unauthorized Access Exception exception is thrown.

→ User-generated Token

You can also use the user-generated token approach to prevent a CSRF attack. This approach enables storing a user-generated token using a HTML hidden field in the user session. Thereafter, for each incoming request from a user, you can verify whether or not the submitted token is valid.

ASP.NET MVC Framework provides a set of helpers that you can use to detect and block any CSRF attacks. This can be done by creating a user-generated token, which is passed between the view and the controller and verified on each request. One of such helper is the `@Html.AntiForgeryToken()` method that allows adding a hidden HTML field in a page that the controller will verify at each request.

Code Snippet 5 shows how to add the `@Html.AntiForgeryToken()` to a form in a view.

Code Snippet 5:

```
<form action="/user/register" method="post">  
    @Html.AntiForgeryToken()  
    ...  
</form>
```

This code will generate an encrypted value for a hidden input and sends back a cookie named, `_RequestVerificationToken`, with the same encrypted value.

Code Snippet 6 shows the generated encrypted value for a hidden input.

Code Snippet 6:

```
<input type="hidden" value="012837udny31w90hjhf7u">
```

In addition, you need to add the `[ValidateAntiForgeryToken]` filter to the action method that processes the posted form.

Code Snippet 7 shows using the `[ValidateAntiForgeryToken]` filter.

Code Snippet 7:

```
[ValidateAntiForgeryToken]  
public ActionResult RegisterUser(RegisterUserModel model)  
{  
    /*Code to register and return an ActionResult object*/  
}
```

In this code, the `[ValidateAntiForgeryToken]` filter checks for the `_RequestVerificationToken` cookie and the `_RequestVerificationToken` form value when a form is posted. If both the values are present, the filter matches. A successful match results in the execution of the `RegisterUser` action. Else, the filter throws an exception.

11.1.2 SQL Injection

SQL injection is a form of attack in which a user posts SQL code to an application. As a result, it creates an SQL statement that will execute with malicious intent. Using a SQL injection, a user can store and update malicious data, access sensitive data, or delete any existing data of the application.

Code Snippet 8 shows a dynamic SQL query that accesses data from the `User` table based on a `userName` form value.

Code Snippet 8:

```
String UserName= context.Request.Form["userName"] ;
String Query = "select * from User where User_name ='" + UserName + "'";
```

In this code, the variable named, `UserName` stores the value of the `userName` field of a submitted form. The `Query` variable constructs a dynamic SQL query by adding the `UserName` variable to retrieve those records from the `User` table whose `User_Name` field matches the value of the `UserName` variable. When a user submits a user name, this code will execute and return all data from the `User` table that matches the user name.

At times, a user who identifies the vulnerabilities for SQL injection in Code Snippet 8, can carry out an attack by submitting a query in the `User name` text field of the form.

Code Snippet 9 shows submitting a query in the `User name` text field of a form.

Code Snippet 9:

```
'; drop table User --
```

On submitting the form, the following query will be generated dynamically.

```
select * from User where User_name = '' ; drop table User --'
```

This code will execute in two phases. In the first phase, the query will attempt to retrieve records from the `User` table. Then, in the next phase, the query will attempt to drop the `User` table. The `--`symbol after the `drop` statement in the query specifies that the remaining part of the query does not required to be executed.

11.1.3 Deferred Validation and Unvalidated Requests

Consider the scenario, where you are developing an online forum that allows user to post queries that contains code snippets. In this application, a Post page that contains four fields to accept the details from a user, such as name, e-mail ID, user query, and any accompanying code snippet. In addition, the Post page contains a **Submit** button, which allows a user to submit the post on the forum.

When a user submits a post after specifying the required details, the request validation feature of the application validates the entire collection of form fields and throws an `HttpRequestValidationException` exception if the code snippet field contains any HTML markups or scripts.

In an ASP.NET MVC application, you can enable deferred validation by configuring it in the `Web.config` file of the application. For this, you need to set the `requestValidationMode` attribute of the `<httpRuntime>` element to 4.5.

Code Snippet 10 allows setting the `requestValidationMode` attribute of the `<httpRuntime>` element.

Code Snippet 10:

```
<httpRuntime requestValidationMode="4.5" />
```

Once you configure deferred validation, you can validate data of a particular field in a form. You can do this, when you expect a field that can contain HTML code or scripts that the request validation process checks for potential XSS attack.

One of the approaches to instruct the request validation process not to validate the data of a particular field is by using the `Unvalidated` property of the `HttpRequest` class.

Code Snippet 11 shows using the `Unvalidated` property to access the content of the code field.

Code Snippet 11:

```
var c = context.Request.Unvalidated.Form["code"];
```

Apart from this, you can also instruct the request validation process not to perform validation at the controller level or at action method level. You can do this by setting the value of the `ValidateInput` attribute to `false`.

Code Snippet 12 shows disabling the validation of form data processed by the `SubmitQuery()` action method.

Code Snippet 12:

```
[HttpPost]
[ValidateInput(false)]
public ActionResult SubmitQuery(QueryModel model)
```

In this code, the value of the `ValidateInput` attribute is set to `false` to disable the validation of form data processed by the `SubmitQuery()` action method.

You can also disable validation to a field for a strongly-typed view by adding the `AllowHtml` attribute to the corresponding property of the model.

Code Snippet 13 shows using the `AllowHtml` attribute.

Code Snippet 13:

```
[AllowHtml]
[Required]
[Display(Name = "Code Snippet")]
public string CodeSnippet { get; set; }
```

In this code, the `AllowHtml` attribute is applied to the `CodeSnippet` property. As a result, when the user adds HTML markups and script in the `CodeSnippet` field and submits the form, the request validation process will not report an error, as it will do for other fields.

11.2 Data Security

All organizations need to handle sensitive data. This data can be either present in storage or may be exchanged between different entities within and outside the organization over a network. Consider a scenario where you are working as a CEO of a company. So, while travelling, you need to access performance appraisal reports of the top management of the company. These reports contain data that are confidential and are stored in the company's server. Such data is often prone to misuse either intentionally with malicious intent or unintentionally. To avoid such misuse, there should be some security mechanism that can ensure confidentiality and integrity of data. One of the commonly used techniques to secure such sensitive data is known as encryption.

11.2.1 Encryption and Decryption

Encryption is a technique that ensures data confidentiality. Encryption converts data in plain text to cipher (secretly coded) text. The opposite process of encryption is called decryption, which is a process that converts the encrypted cipher text back to the original plain text.

Figure 11.2 shows the process of encryption and decryption of a password as an example.

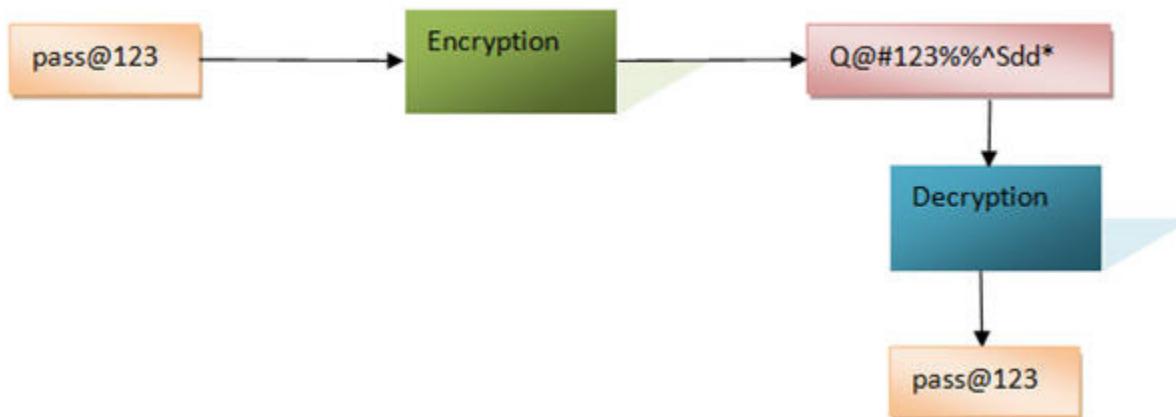


Figure 11.2: Encryption and Decryption

In this figure, the plain text, pass@123 is encrypted to a cipher text. The cipher text is decrypted back to the original plain text.

→ Types of Encryption and Decryption

There are two types of encryption and decryption, such as symmetric and asymmetric.

Symmetric encryption or secret key encryption, uses a single key, known as the secret key both to encrypt and decrypt data. The following steps outline an example usage of symmetric encryption:

1. User A uses a secret key to encrypt a plain text to cipher text.
2. User A shares the cipher text and the secret key with User B.

- User B uses the secret key to decrypt the cipher text back to the original plain text.

Figure 11.3 shows the symmetric encryption and decryption process.

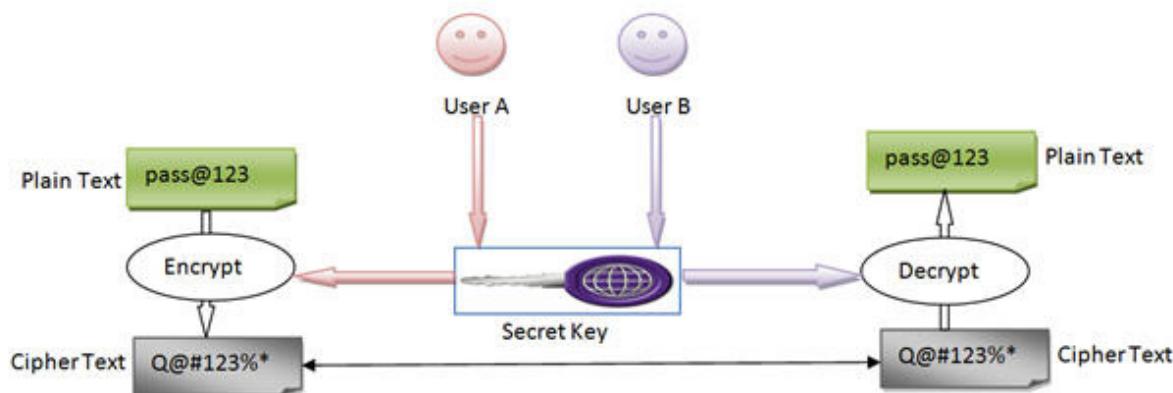


Figure 11.3: Symmetric Encryption and Decryption

Note - To make symmetric encryption more secure, an Initialization Vector (IV) can be used with the secret key. An IV is a random number that generates different sequence of encrypted text for identical sequence of text present in the plain text. When you use an IV with a key to symmetrically encrypt data, you will need the same IV and key to decrypt data.

On the other hand, the asymmetric encryption uses a pair of public and private key to encrypt and decrypt data. The following steps outline an example usage of asymmetric encryption:

- User A generates a public and private key pair.
- User A shares the public key with User B.
- User B uses the public key to encrypt a plain text to cipher text.
- User A uses the private key to decrypt the cipher text back to the original plain text.

Figure 11.4 shows the asymmetric encryption and decryption process.

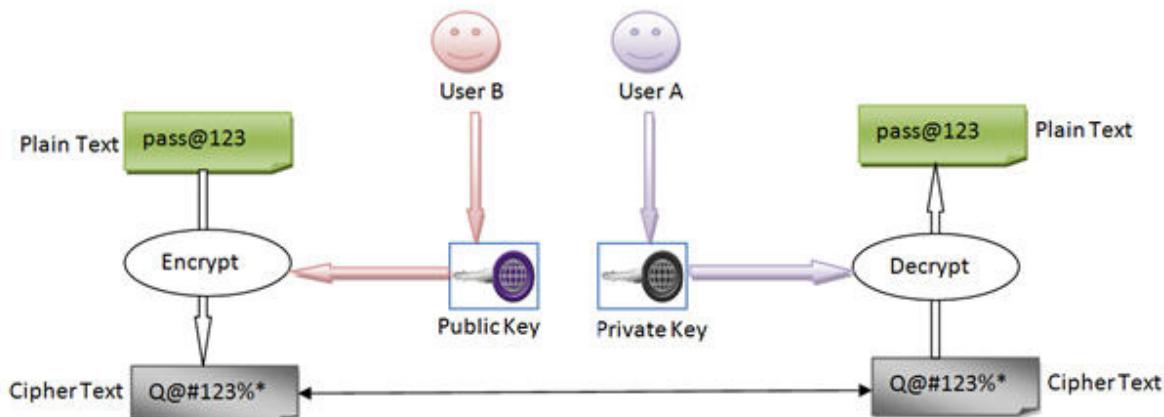


Figure 11.4: Asymmetric Encryption and Decryption

→ Using Symmetric Encryption

You need to use one of the symmetric encryption implementation classes of the .NET Framework to perform symmetric encryption. The first step to perform symmetric encryption is to create the symmetric key.

When you use the default constructor of the symmetric encryption classes, such as `RijndaelManaged` and `AesManaged`, a key and IV is automatically generated. The generated key and the IV can be accessed as byte arrays using the `Key` and `IV` properties of the encryption class.

Code Snippet 14 shows creating a symmetric key and IV using the `RijndaelManaged` class.

Code Snippet 14:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
...
RijndaelManaged symAlgo = new RijndaelManaged();
Console.WriteLine("Generated key: {0}, \nGenerated IV: {1}", Encoding.Default.GetString(symAlgo.Key), Encoding.Default.GetString(symAlgo.IV));
```

This code uses the default constructor of the `RijndaelManaged` class to generate a symmetric key and IV. The Key and IV properties are accessed and printed as strings using the default encoding to the console.

Figure 11.5 shows the output of Code Snippet 14.

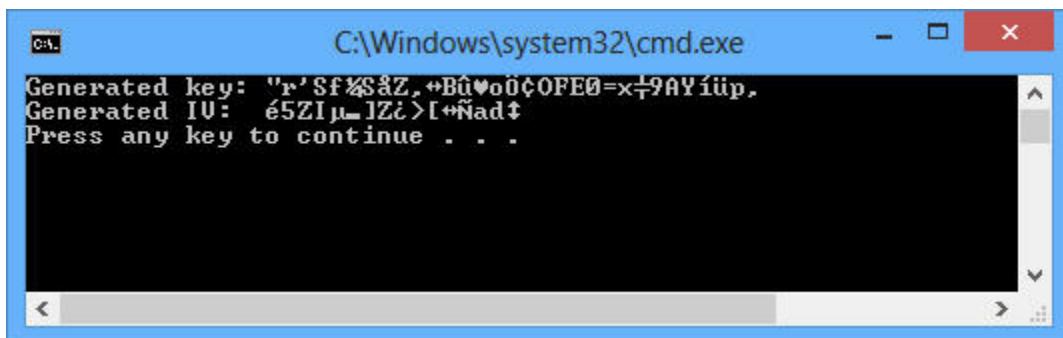


Figure 11.5: Output of Code Snippet 14

The symmetric encryption classes, such as `RijndaelManaged` also provide the `GenerateKey()` and `GenerateIV()` methods that you can use to generate keys and IVs, as shown in Code Snippet 15.

Code Snippet 15:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
...
RijndaelManagedsymAlgo = new RijndaelManaged();
    RijndaelManagedsymAlgo.GenerateKey();
    RijndaelManagedsymAlgo.GenerateIV();
    byte[] generatedKey = RijndaelManagedsymAlgo.Key;
    byte[] generatedIV = RijndaelManagedsymAlgo.IV;
    Console.WriteLine("Generated key through GenerateKey(): {0}, \nGenerated
IV through GenerateIV(): {1}",
    Encoding.Default.GetString(generatedKey),
    Encoding.Default.GetString(generatedIV));
```

This code creates a `RijndaelManaged` object and then, calls the `GenerateKey()` and `GenerateIV()` methods to generate a key and an IV. The Key and the IV properties are then, accessed and printed as strings using the default encoding to the console.

Figure 11.6 shows the output of Code Snippet 15.

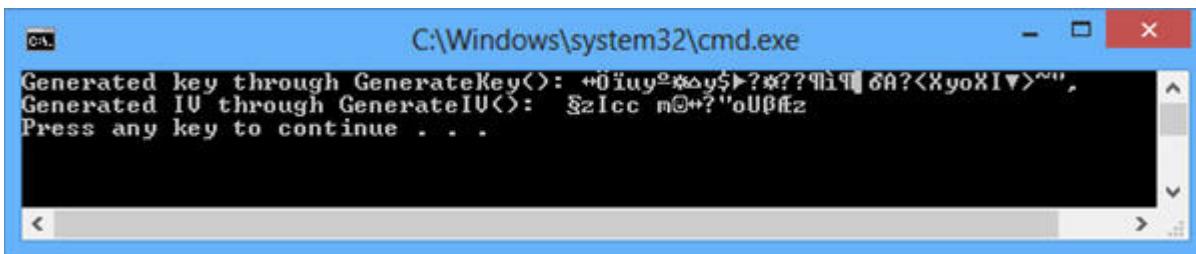


Figure 11.6: Output of Code Snippet 15

The symmetric encryption classes of the ASP.NET MVC Framework provides the `CreateEncryptor()` method that returns an object of the `ICryptoTransform` interface. The `ICryptoTransform` object is responsible for transforming the data based on the algorithm of the encryption class. Once you have obtained an `ICryptoTransform` object, you can use the `CryptoStream` class to perform encryption. The `CryptoStream` class acts as a wrapper of a stream-derived class, such as `FileStream`, `MemoryStream`, and `NetworkStream`. A `CryptoStream` object operates in one of the two modes defined by the `CryptoStreamMode`

enumeration. First one is Write mode that allows writing operation on the underlying stream and you can use this mode to perform encryption. The second one is the Readmode that allows reading operation on the underlying stream. You can use this mode to perform decryption.

You can create a `CryptoStream` object by calling the constructor that accepts the following three parameters:

- The underlying stream object
- The `ICryptoTransform` object
- The mode defined by the `CryptoStreamMode` enumeration

After creating the `CryptoStream` object, you can call the `Write()` method to write the encrypted data to the underlying stream.

Code Snippet 16 encrypts data using the `RijndaelManaged` class and writes the encrypted data to a file.

Code Snippet 16:

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class SymmetricEncryptionDemo
{
    static void EncryptData(String plainText, RijndaelManaged algo)
    {
        byte[] plaindataArray=ASCIIEncoding.ASCII.GetBytes(plainText);
        ICryptoTransform transform=algo.CreateEncryptor();
        using (var fileStream=new FileStream("D:\\CipherText.txt", FileMode.
OpenOrCreate, FileAccess.Write))
        {
            using (var cryptoStream=new CryptoStream(fileStream, transform,
CryptoStreamMode.Write))
            {
                cryptoStream.Write(plaindataArray, 0, plaindataArray.
GetLength(0));
                Console.WriteLine("Encrypted data written to: D:\\CipherText.txt");
            }
        }
    }
}
```

```
}

static void Main()
{
    RijndaelManaged symAlgo = new RijndaelManaged();
    Console.WriteLine("Enter data to encrypt.");
    string dataToEncrypt = Console.ReadLine();
    EncryptData(dataToEncrypt, symAlgo);
}
```

In this code, the `Main()` method creates a `RijndaelManaged` object and passes it along with the data to encrypt the `EncryptData()` method. In the `EncryptData()` method, the call to the `CreateEncryptor()` method creates the `ICryptoTransform` object. Then, a `FileStream` object is created to write the encrypted text to the `CipherText.txt` file. Next, the `CryptoStream` object is created and its `Write()` method is called.

Figure 11.7 shows the output of Code Snippet 16.

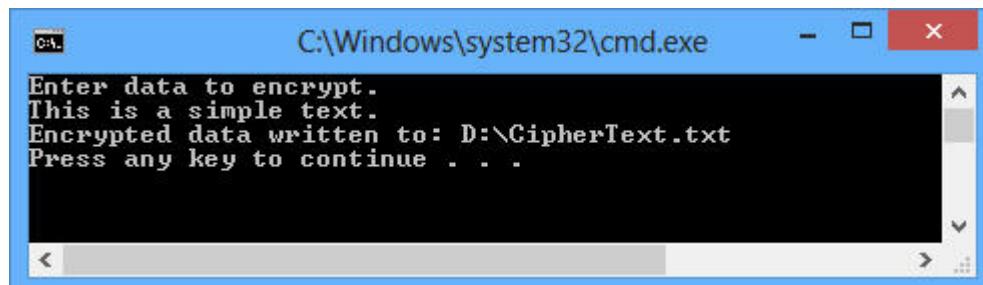


Figure 11.7: Output of Code Snippet 16

On the other hand, to decrypt data, you need to use the same symmetric encryption class, key and IV used for encrypting the data. You call the `CreateDecryptor()` method to obtain a `ICryptoTransform` object that will perform the transformation. You then, need to create the `CryptoStream` object in Read mode and initialize a `StreamReader` object with the `CryptoStream` object. Finally, you need to call the `ReadToEnd()` method of the `StreamReader` that returns the decrypted text as a string.

Code Snippet 17 shows creating a program that performs both encryption and decryption.

Code Snippet 17:

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class SymmetricEncryptionDemo
{
    static void EncryptData(String plainText, RijndaelManaged algo)
    {
        byte[] plaindataArray = ASCIIEncoding.ASCII.GetBytes(plainText);
        ICryptoTransform transform = algo.CreateEncryptor();
        using (var fileStream = new FileStream("D:\\CipherText.txt", FileMode.
OpenOrCreate, FileAccess.Write))
        {
            using (var cryptoStream = new CryptoStream(fileStream, transform,
CryptoStreamMode.Write))
            {
                cryptoStream.Write(plaindataArray, 0, plaindataArray.GetLength(0));
                Console.WriteLine("Encrypted data written to: D:\\CipherText.txt");
            }
        }
    }
    static void DecryptData(RijndaelManaged algo)
    {
        ICryptoTransform transform = algo.CreateDecryptor();
        using (var fileStream = new FileStream("D:\\CipherText.txt", FileMode.
Open, FileAccess.Read))
        {
            using (CryptoStream cryptoStream = new CryptoStream(fileStream,
transform, CryptoStreamMode.Read))
        }
    }
}
```

```
}

using (var streamReader = new StreamReader(cryptoStream))
{
    string decryptedData = streamReader.ReadToEnd();
    Console.WriteLine("Decrypted data: \n{0}", decryptedData);
}

}

}

static void Main()
{
    RijndaelManaged symAlgo = new RijndaelManaged();
    Console.WriteLine("Enter data to encrypt.");
    string dataToEncrypt = Console.ReadLine();
    EncryptData(dataToEncrypt, symAlgo);
    DecryptData(symAlgo);
}

}
```

In this code,

- The Main() method creates a RijndaelManaged object and passes it along with the data to encrypt the EncryptData() method. The encrypted data is saved to the CipherText.txt file.
 - The Main() method calls the DecryptData() method passing the same RijndaelManaged object created for encryption.
 - The DecryptData() method creates the ICryptoTransform object and uses a FileStream object to read the encrypted data from the file.
 - The CryptoStream object is created in the Read mode and initialized with the FileStream and ICryptoTransform objects.
 - A StreamReader object is created by passing the CryptoStream object to the constructor.
 - The ReadToEnd() method of the StreamReader object is called.

Figure 11.8 shows the decrypted text returned by the `ReadToEnd()` method.

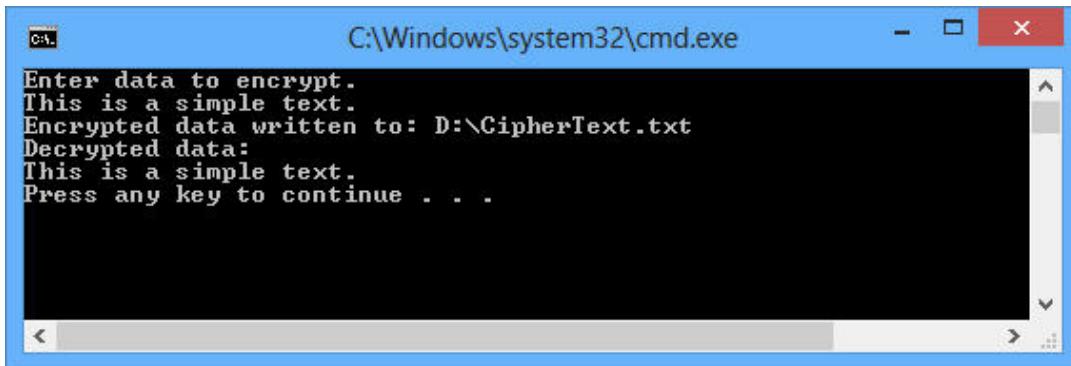


Figure 11.8: Decrypted Text Returned by `ReadToEnd()` Method

→ Using Asymmetric Encryption

The `System.Security.Cryptography` namespace provides the `RSACryptoServiceProvider` class that you can use to perform asymmetric encryption.

When you call the default constructor of the `RSACryptoServiceProvider` class, a new public-private key pair is automatically generated. After you create a new instance of this class, you can export the key information by using one of the following methods:

- `ToXMLString()`: Returns an XML representation of the key information.
- `ExportParameters()`: Returns an `RSAParameters` structure that holds the key information.

Both the `ToXMLString()` and `ExportParameters()` methods accept a Boolean value. A false value indicates that the method should return only the public key information, while a true value indicates that the method should return information of both the public and private keys.

Code Snippet 18 shows creating and initializing an `RSACryptoServiceProvider` object and then, exports the public key in XML format.

Code Snippet 18:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
string publicKey = rSAKeyGenerator.ToXmlString(false);
```

Code Snippet 19 shows creating and initializing an RSACryptoServiceProvider object.

Code Snippet 19:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(true);
```

This code creates and initializes an RSACryptoServiceProvider object and then, exports both the public and private keys as an RSAParameters structure.

Now, to encrypt data, you need to create a new instance of the RSACryptoServiceProvider class and call the ImportParameters() method to initialize the instance with the public key information exported to an RSAParameters structure.

Code Snippet 20 shows initializing an RSACryptoServiceProvider object with the public key exported to an RSAParameters structure.

Code Snippet 20:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
...
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(false);
RSACryptoServiceProvider rsaEncryptor = new RSACryptoServiceProvider();
rsaEncryptor.ImportParameters(rSAKeyInfo);
```

If the public key information is exported to XML format, you need to call the FromXmlString() method to initialize the RSACryptoServiceProvider object with the public key.

Code Snippet 21 shows initializing the `RSACryptoServiceProvider` object with the public key.

Code Snippet 21:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
string publicKey = rSAKeyGenerator.ToXmlString(false);
RSACryptoServiceProvider rsaEncryptor = new RSACryptoServiceProvider();
rsaEncryptor.FromXmlString(publicKey);
```

Once you have initialized the `RSACryptoServiceProvider` object with the public key, you can encrypt data by using the `Encrypt()` method of the `RSACryptoServiceProvider` class. The `Encrypt()` method accepts the two parameters, `byte` array of the data to encrypt and a Boolean value that indicates whether or not to perform encryption.

The `Encrypt()` method after performing encryption returns a `byte` array of the encrypted text.

Code Snippet 22 shows returning a `byte` array of the encrypted text.

Code Snippet 22:

```
byte[] plainbytes = new UnicodeEncoding().GetBytes("Plaintext to
encrypt.");
byte[] cipherbytes = rsaEncryptor.Encrypt(plainbytes, true);
```

To decrypt data, you need to initialize an `RSACryptoServiceProvider` object using the private key of the key pair whose public key was used for encryption.

Code Snippet 23 shows how to initialize an `RSACryptoServiceProvider` object with the private key exported to an `RSAParameters` structure.

Code Snippet 23:

```
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(true);
RSACryptoServiceProvider rsaDecryptor = new RSACryptoServiceProvider();
rsaDecryptor.ImportParameters(rSAKeyInfo);
```

Code Snippet 24 shows how to initialize an `RSACryptoServiceProvider` object with the private key exported to XML format.

Code Snippet 24:

```
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
string keyPair = rSAKeyGenerator.ToXmlString(true);
RSACryptoServiceProvider rsaDecryptor = new RSACryptoServiceProvider();
rsaDecryptor.FromXmlString(keyPair);
```

When the `RSACryptoServiceProvider` object is initialized with the private key, you can decrypt data by using the `Decrypt()` method of the `RSACryptoServiceProvider` class. The `Decrypt()` method accepts the two parameters, a byte array of the encrypted data and a Boolean value that indicates whether or not to perform encryption.

Code Snippet 25 shows the byte array of the original data returned by the `Decrypt()` method.

Code Snippet 25:

```
byte[] plainbytes = rsaDecryptor.Decrypt(cipherbytes, false);
```

Code Snippet 26 shows a program that performs asymmetric encryption and decryption.

Code Snippet 26:

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class AsymmetricEncryptionDemo
{
    static byte[] EncryptData(string plainText, RSAParameters rsaParameters)
    {
        byte[] plainTextArray = new UnicodeEncoding().GetBytes(plainText);
        RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
        RSA.ImportParameters(rsaParameters);
        byte[] encryptedData = RSA.Encrypt(plainTextArray, true);
        return encryptedData;
    }
    static byte[] DecryptData(byte[] encryptedData, RSAParameters
```

```

rsaParameters)

{
    RSACryptoServiceProvider RSA = new
    RSACryptoServiceProvider();
    RSA.ImportParameters(rsaParameters);
    byte[] decryptedData = RSA.Decrypt(encryptedData, true);
    return decryptedData;
}

static void Main(string[] args)
{
    Console.WriteLine("Enter text to encrypt:");
    String inputText = Console.ReadLine();
    RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
    RSAParameters RSAParam = RSA.ExportParameters(false);
    byte[] encryptedData = EncryptData(inputText, RSAParam);
    string encryptedString = Encoding.Default.GetString(encryptedData);
    Console.WriteLine("\nEncrypted data \n{0}", encryptedString);
    byte[] decryptedData = DecryptData(encryptedData,
        RSA.ExportParameters(true));
    String decryptedString = new UnicodeEncoding().GetString(decryptedData);
    Console.WriteLine("\nDecrypted data \n{0}", decryptedString);
}
}

```

In this code,

- The `Main()` method creates a `RSACryptoServiceProvider` object and exports the public key as a `RSAParameters` structure.
- The `EncryptData()` method is called passing the user entered plain text and the `RSAParameters` object. The `EncryptData()` method uses the exported public key to encrypt the data and returns the encrypted data as a byte array.
- The `Main()` method then, exports both the public and private key of the `RSACryptoServiceProvider` object into a second `RSAParameters` object.
- The `DecryptData()` method is called passing the encrypted byte array and the `RSAParameters` object.

- Finally, the `DecryptData()` method performs the decryption and returns the original plain text as a string.

Figure 11.9 shows the output of Code Snippet 26.

```

Enter text to encrypt:
This is a simple text.

Encrypted data
u1sa1z'ti!G+i%CI".OND@%`,_satyl+<dUif8&li+i@0/-z@?'>>T@20@o.^ZS+o@n@_0@0*E^"UqH@p
<400.é'@-C3@í_@-VEYd!po=,"_@+@aç@Y@'@ç1@p-T@# @Ko

Decrypted data
This is a simple text.
Press any key to continue . . .

```

Figure 11.9: Output of Code Snippet 26

11.2.2 Salting and Hashing

Consider a scenario, where you are creating an ASP.NET MVC application that stores user credentials in the form of user name and password in a database. Though you have implemented security to prevent the data of the application from malicious attacks, the user credentials stored in the database might be compromised if an attacker gains direct access to the database. So, you need to ensure that the user credentials are secure in the database. For this, you need to use an additional security measures in the application.

To use such additional security on the data stored in a database, you can use two techniques, known as hashing and salting.

Hashing is a technique that allows generating a unique hash value of data by using a hashing algorithm. Once you have generated a unique hash value for any data, this data cannot be converted back to the original form. The process of hashing in order to protect passwords in a database can be better understood by a series of steps. In this process, first, the application generates a hash value for a user's password and stores it to the database. Next, the application receives a login request comprising a user name and password. Then, the application generates a new hash value for the received password. Finally, the application compares the newly generated hash value with the existing value stored in the database. If both the hash values match, the password is assumed to be correct.

On the other hand, salting is another security technique that you can apply on hashing. Salting allows creating and adding a random string to the input data before generating a hash value. The process of hashing in combination with salting to protect passwords stored in the database can be better understood by a series of steps. In this process, at first, the application adds a random salt to the user's password and generates a hash value and the application stores both the hash value and the salt to the database. Then, the application receives a login request with a user name and password. Next, the application retrieves the salt, applies it to the received password, and generates a new hash value. Finally, the application

compares both the new hash value and the existing one stored in the database. If both the hash values match, the password is assumed to be correct.

11.2.3 Digital Signature

You can use digital signature to authenticate the identity of a sender of some kind of data. This type of signature also ensures that the data has not been tampered while in transit. By using digital signature, a user sending a signed data cannot later deny his or her ownership of the data.

The process of using digital signature requires a series of operations to be performed. At first, a sender computes a hash value from the data being sent. Then, the sender encrypts the hash value with the private key of an asymmetric key pair. The encrypted hash value is the digital signature. Next, the sender sends the data and the digital signature to the receiver. Next, the receiver computes a hash from the received data. Finally, the receiver decrypts the signature using the public key of the sender and then, compares the hash values for authenticity. If the decrypted hash value matches with the recipient's computed hash value, the signature is valid and the data is ensured to be intact.

To understand how digital signatures work, you first need to generate a digital signature for some data. You can then, verify the signature.

To generate the asymmetric keys whose private key can be used to sign the data, you need to use the RSACryptoServiceProvider class. Then, you need to store the asymmetric keys using a secured mechanism. For this, you can use a key container that is a logical structure that enables securely storing asymmetric keys. Then, the CspParameters class can be used to create a key container and to add and remove keys to and from the container.

To create a key container for an RSACryptoServiceProvider object, you first need to use the default constructor of the CspParameters class to create a key container instance. Then, you need to set the container name using the KeyContainerName property of the CspParameters class.

Code Snippet 27 shows using the KeyContainerName property of the CspParameters class.

Code Snippet 27:

```
CspParameters param = new CspParameters();
param.KeyContainerName = "SignatureContainer121";
```

Now, you need to store the key pair of the RSACryptoServiceProvider object in the key container. For this, you need to pass the CspParameters object to the constructor while creating the RSACryptoServiceProvider object. Then, you need to set the PersistKeyInCsp property of the RSACryptoServiceProvider object to true.

Code Snippet 28 shows how to set the `PersistKeyInCsp` property.

Code Snippet 28:

```
using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(param))
{
    rsa.PersistKeyInCsp = true;
    ...
}
```

In this code, an `RSACryptoServiceProvider` object is initialized with a `CspParameters` object and then, the key pair is stored in the `CspParameters` object.

Thereafter, you can generate the signature by using the `SignData()` method. This method accepts the data to sign in a byte array and the hash algorithm is used to create the hash value. You can pass the `SHA256` string value to use the hash algorithm using the `SignData()` method. The `SignData()` method after generating the digital signature, returns the signature in a byte array.

Code Snippet 29 shows how to use the `SignData()` method.

Code Snippet 29:

```
byte[] byteData = Encoding.Unicode.GetBytes("Data to Sign.");
byte[] byteSignature = rsa.SignData(byteData, "SHA256");
```

In this code, the `SignData()` generates a digital signature and returns the signature in a byte array.

Once you have created the digital signature, you can verify it. For that, you need to first create a `CspParameters` object and use the same key container name that you used while creating the signature. This will ensure that when you later create the `RSACryptoServiceProvider` object, it will be initialized with the same key pair that was used to generate the signature.

Code Snippet 30 shows creating a `CspParameters` object.

Code Snippet 30:

```
CspParameters param = new CspParameters();
param.KeyContainerName = "SignatureContainer101";
```

After creating a `CspParameters` object, you can create the `RSACryptoServiceProvider` object initialized with the `CspParameters` object. You can then, call the `VerifyData()` method. This method accepts three different types of parameters, such as the byte array of the data that needs to be verified against the signature, the hash algorithm that is used to generate the signature, and the byte array of the generated signature.

The `VerifyData()` method returns a Boolean value `true` if the signature is successfully verified, otherwise `false`.

Code Snippet 31 shows using the VerifyData() method that returns a Boolean value true.

Code Snippet 31:

```
bool isSuccess = false;  
  
using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(param))  
{  
  
    isSuccess = rsa.VerifyData(byteData, "SHA256", byteSignature);  
}
```

This code creates an RSACryptoServiceProvider object and then, calls the VerifyData() method to verify the signature.

11.3 Check Your Progress

1. Which of the following exception is thrown by the request validation feature when a user attempts to submit HTML or JavaScript code to carry out an XSS attack through the body, header, or cookies of the request?

(A)	HttpResponseValidationException		
(B)	UnauthorizedAccessException		
(C)	UnauthorizedRequestAccessException		
(D)	HttpRequestValidationException		
(E)	HttpRequestResponseValidationException		

(A)	A	(C)	D
(B)	E	(D)	B

2. Which of the following methods allows adding a hidden HTML field in a page that the controller verifies at each request?

(A)	GenerateKey()		
(B)	ValidateInput()		
(C)	@Html.AntiForgeryToken()		
(D)	CreateEncryptor()		
(E)	ExportParameters()		

(A)	B	(C)	D
(B)	C	(D)	A

3. Which of the following method allows you to initialize an instance of the `RSACryptoServiceProvider` class with the public key information exported to an `RSAParameters` structure?

(A)	ExportParameters()		
(B)	ImportParameters()		
(C)	Encrypt()		
(D)	SignData()		

(E)	VerifyData()
-----	--------------

(A)	B	(C)	A
(B)	C	(D)	E

4. Which of the following methods accepts the data to sign in a byte array and the hash algorithm to use to create the hash value?

(A)	SignData()
(B)	EncryptData()
(C)	FromXmlString()
(D)	ImportParameters()
(E)	CreateDecryptor()

(A)	B	(C)	D
(B)	E	(D)	A

5. Which of the following properties will you use in the action method to retrieve the host name of your application?

(A)	KeyContainerName
(B)	PersistKeyInCsp
(C)	Unvalidated
(D)	Request.UrlReferrer.Host
(E)	Request.Url.Host

(A)	B	(C)	A
(B)	C	(D)	E

11.3.1 Answers

(1)	C
(2)	B
(3)	A
(4)	D
(5)	D

Summary

- HTML encoding is a process that converts potentially unsafe characters to their HTML-encoded equivalent.
- The ASP.NET MVC Framework provides a request validation feature that examines an HTTP request to check and prevent potentially malicious content.
- The user-generated token approach enables storing a user-generated token using a HTML hidden field in the user session.
- SQL injection is a form of attack in which a user posts SQL code to an application, thus it creates an SQL statement that will execute with malicious intent.
- Encryption is a technique that ensures data confidentiality by converting data in plain text to cipher (secretly coded) text.
- Hashing is a technique that allows generating a unique hash value of data by using a hashing algorithm.
- Salting is a security technique that you can apply on hashing to create and add a random string to the input data before generating a hash value.



Visit
Frequently Asked Questions
@

Session - 12

Globalization

Welcome to the Session, **Globalization**. Globalization is a process of designing and developing applications that can be used for multiple cultures. Globalization combines the processes of internationalization and localization. The ASP.NET MVC Framework supports various techniques, that enables you develop application such as globalized applications.

In an ASP.NET MVC application, you can also implement localization to display content in languages that a user prefers. You can achieve this by separating the language-related and the culture-related content from the application code.

In this Session, you will learn to:

- ➔ Define and describe how to implement internationalization
- ➔ Define and describe how to implement localization

12.1 Implementing Internationalization

At recent times, most of the organizations do business in the global marketplace. So these types of organization must design applications to accommodate users from a wide variety of cultures. Users in different parts of the world use different languages. Therefore, you need to enable your Web applications to represent information in the users' native language and formats. This functionality can be provided by implementing internationalization.

While developing a global application, you should ensure that the application should adapt to multiple languages and cultures without any changes in the code structure of the application. You can do this with the help of the internationalization process.

The ASP.NET MVC Framework allows you to develop such multicultural applications. You can develop an ASP.NET MVC application to retrieve information about a particular culture and region to which a user belongs. Based on this information, you can format and present your application data in the desired language.

12.1.1 Culture Code

In order to implement internationalization in an ASP.NET MVC application, you need to understand about the culture code that the .NET Framework uses. The .NET Framework uses the following syntax to represent a culture code:

Syntax:

<languagecode>-<country/regioncode>

where,

<languagecode>: Is a lowercase two-letter code for a language.

<country/regioncode>: Is an uppercase two-letter code for a country or a region. For example, the culture name en-US represents the English language of the US. However, the culture name en-GB represents the English language of the UK.

In an ASP.NET MVC application, you can access all the supported culture of the .NET Framework using the `CultureInfo` class of the `System.Globalization` namespace, as shown in Code Snippet 1.

Code Snippet 1:

```
ArrayList cultureNameList = new ArrayList();
CultureInfo[] cInfo = CultureInfo.GetCultures(CultureTypes.AllCultures);
foreach (CultureInfo cultures in cInfo)
{
    cultureNameList.Add(String.Format("Culture Name {0} :
    DisplayName {1}", cultures.Name, cultures.DisplayName));
}
```

This code calls the `CultureInfo.GetCultures()` method passing a `CultureTypes.AllCultures` enumeration member. Next, this code accesses the `CultureInfo.Name` and `CultureInfo.DisplayName` properties of all the supported cultures and adds the information as a formatted string to an `ArrayList`.

12.1.2 Setting Cultures

Before you configure internationalization in an ASP.NET MVC application, you need to understand about the `Culture` and `UICulture` properties of the `Page` class. Both these properties are part of each ASP.NET page of the application. The `Culture` property is a string that determines the results of culture-dependent functions, such as formatting of date and time, number, and currency. The `UICulture` property, which is also a string, determines the resource file that contains locale specific content for a page.

While configuring internationalization in an application, you might often need to consider assigning different values for the `Culture` and `UICulture` properties. For example, consider a scenario where you are developing an online shopping store application. In this application, you want to display the content of the Web page in English (United States), English (United Kingdom), and French (France) based on user preferences. You can achieve this by using the `UICulture` property.

In an ASP.NET MVC, you can set and access the current culture and UI culture of a page from the request processing thread using the `Thread.CurrentThread.CurrentCulture` and `Thread.CurrentThread.CurrentUICulture` properties.

The .NET Framework examines both these properties before rendering culture-dependent functions. You can explicitly set a specific culture, such as `fr-FR`, `en-UK`, or `es-ES` as the value of the `CurrentCulture` and `CurrentUICulture` properties.

Code Snippet 2 shows explicitly setting the `CurrentCulture` property of the thread executing the `Index()` action method of the Home controller:

Code Snippet 2:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        Thread.CurrentThread.CurrentCulture = new System.Globalization.CultureInfo("fr-FR");
        ViewBag.CultureName = System.Globalization.CultureInfo.CurrentCulture.Name;
        ViewBag.Message = DateTime.Now.ToString("yyyy-MM-dd");
        return View();
    }
}
```

This code sets the `CurrentCulture` property to a `CultureInfo` object initialized with the `fr-FR` culture name. The code then, adds the name of the current culture and the current date to a `ViewBag` object before it returns the view.

In the corresponding `Index.cshtml` view, you can access and display the `ViewBag` messages. Code Snippet 3 shows how to access and display the `ViewBag` messages.

Code Snippet 3:

```
<h2>@ViewBag.CultureName @ViewBag.Message</h2>
```

When the `Index.cshtml` view is accessed in a browser, it displays the `fr-FR` as the culture name and the date in French.

Figure 12.1 shows the `Index.cshtml` view in French.

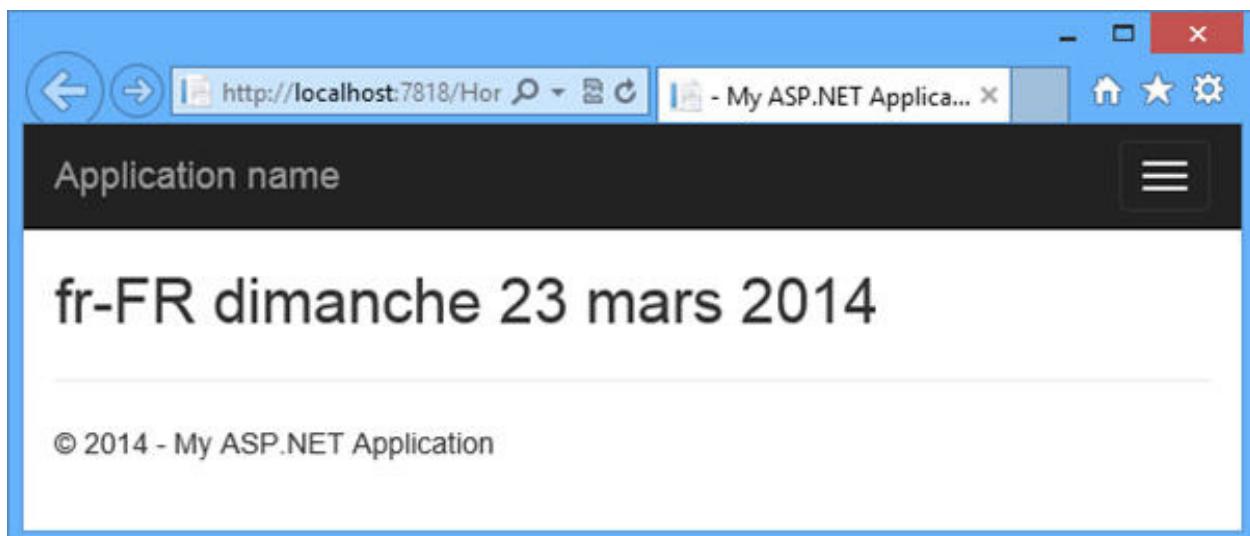


Figure 12.1: Current Culture Name and Date

12.1.3 Using the Web.config File

In an ASP.NET MVC application, you can use the `Web.config` file to use the language that is set as the default language in the browser that sends requests to the application. For that you need to know how to set the preferred language of the browser. Then, you need to know how to configure the `Web.config` file to use the preferred language.

→ Configuring Preferred Language of Browser

To set the preferred language in a browser, you need to perform the following steps:

1. Open **Internet Explorer** (IE).
2. Press the **Alt+X** keys to open the **Tools** menu.
3. Click **Internet Options** in the **Tools** menu. The **Internet Options** dialog box appears.

4. Click **Languages** in the **Internet Options** dialog box. The **Language Preference** dialog box appears.
5. Click **Set Language Preferences** in the **Language Preference** dialog box. The **Add Languages** window appears.
6. Locate and select **French** in the **Add Languages** window. Figure 12.2 shows selecting **French** in the **Add Languages** window.

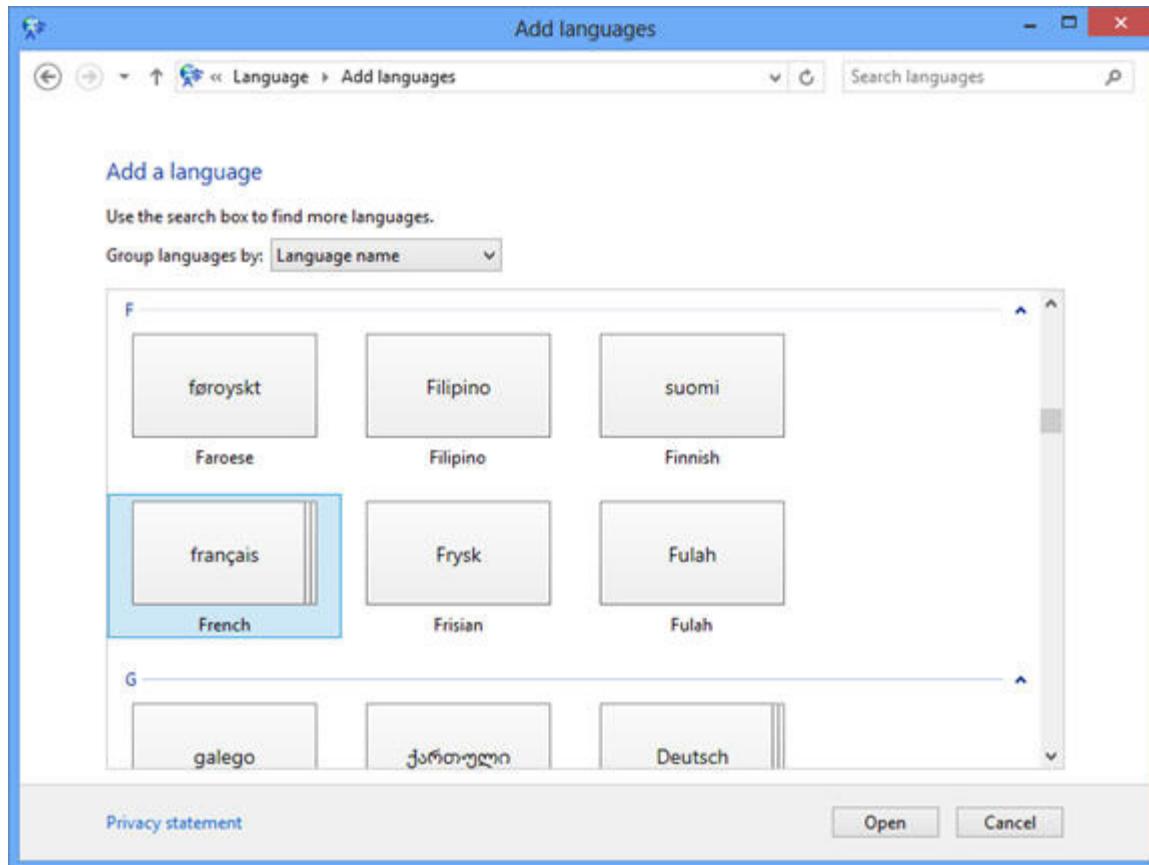


Figure 12.2: Selecting the French Language

7. Click **Open**. The **Regional variants** dialog box is displayed.
8. Select **French (France)** in the **Regional variants** dialog box.

Figure 12.3 shows selecting **French (France)** in the **Regional variants** dialog box.

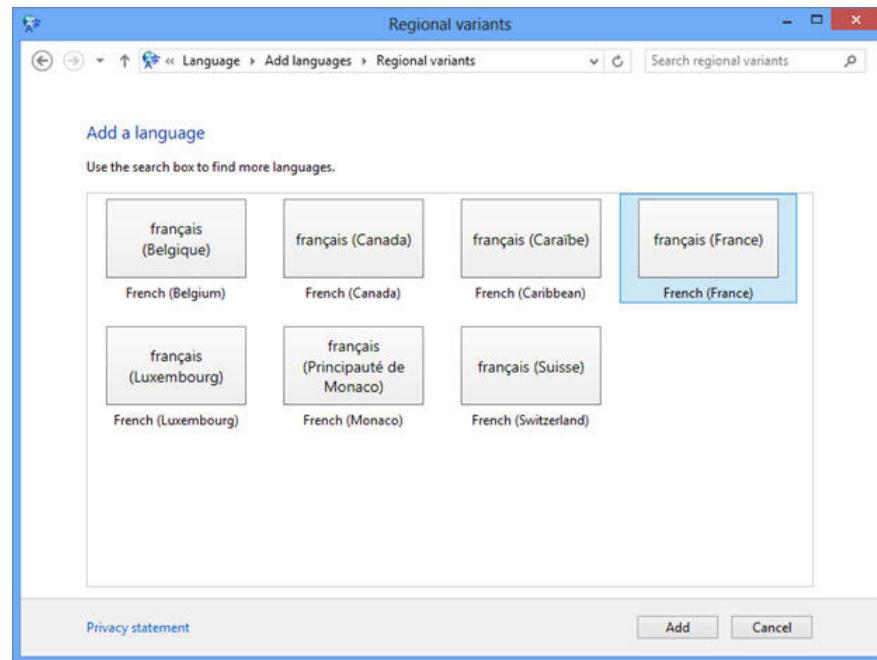


Figure 12.3: Selecting the French(France) Language

- Click **Add**. The **Language** window displays the newly added language. Figure 12.4 shows the newly added **francais (France)** language.

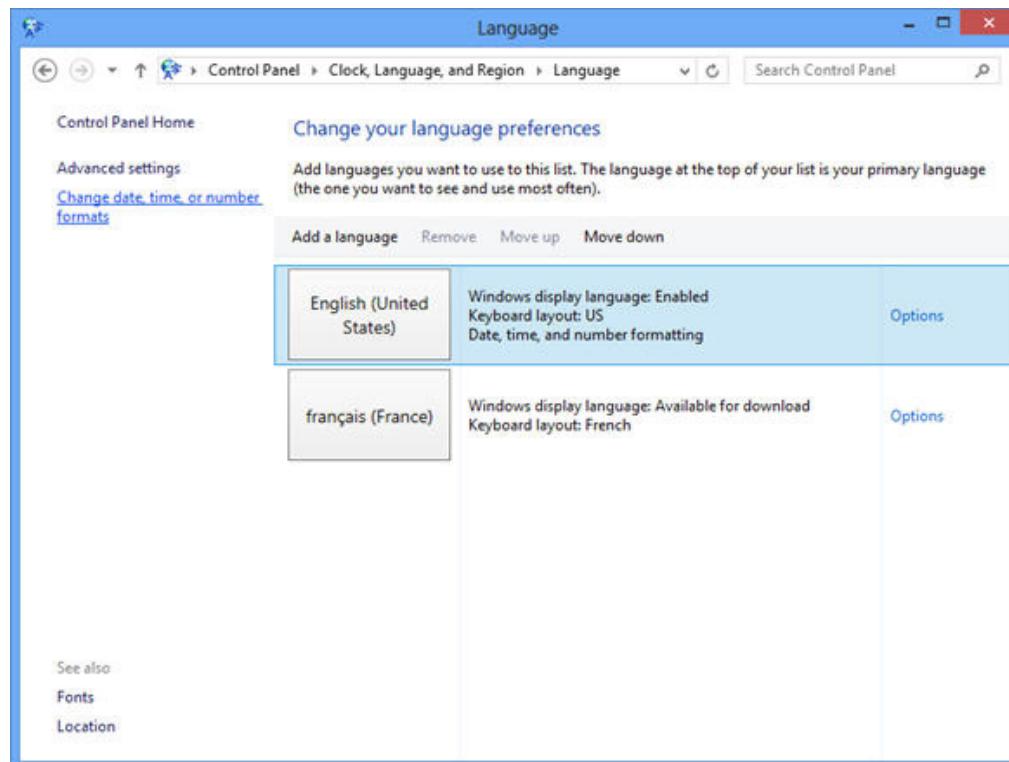


Figure 12.4: Newly Added francais (France) Language

10. Select **francais (France)** and click **Move up** to make French (France) as the first preferred language of the browser.
11. The **Language** window displays **francais (France)** at the top of the language list. Figure 12.5 shows **francais (France)** at the top of the language list.

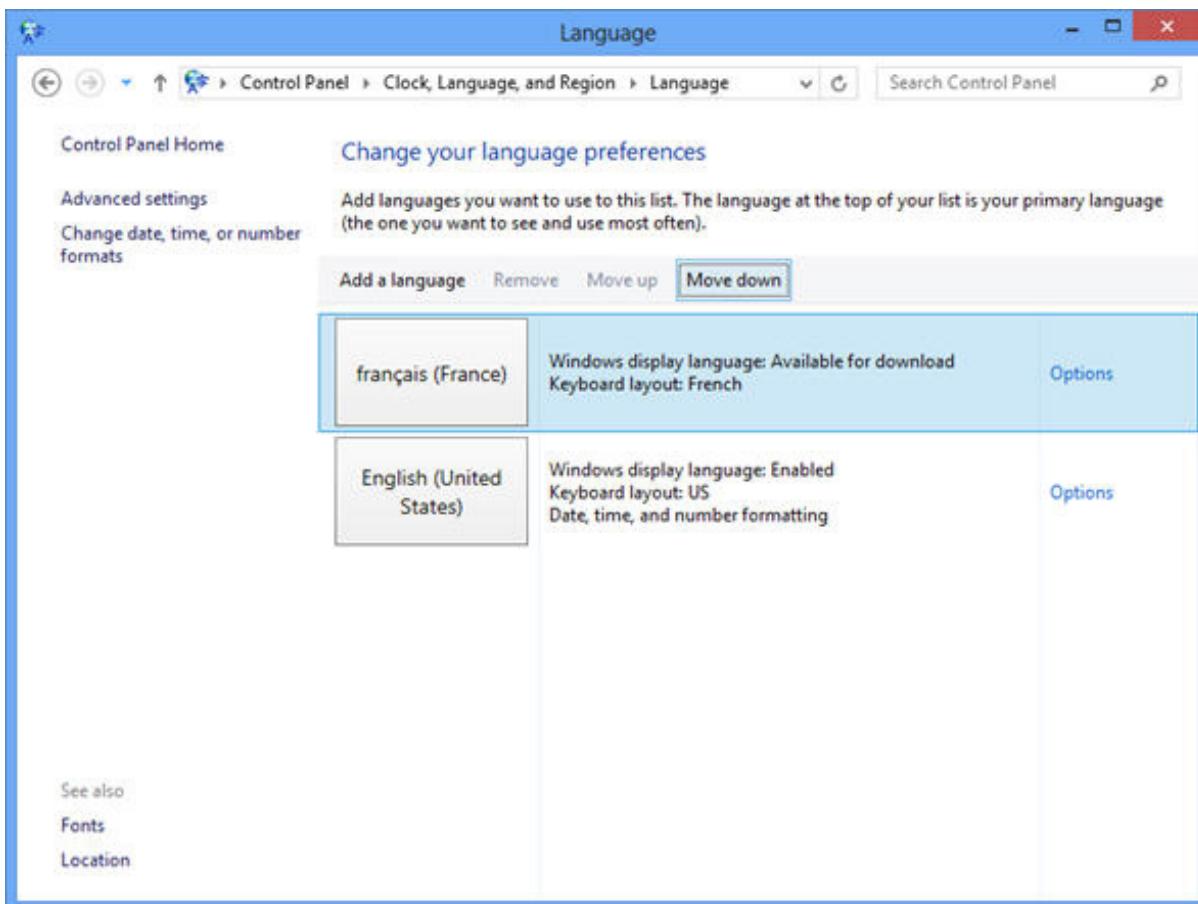


Figure 12.5: Displaying **francais (France)** as Preferred Language

12. Close the **Language** window.
13. Close the **Internet Options** dialog box.
14. Click **OK** in the **Language Preferences** dialog box.
15. Click **OK** in the **Internet Options** dialog box.

→ Configuring the **Web.config** File

In ASP.NET MVC application, you can configure the **Web.config** file to use the browser preferred language by adding the `<globalization>` element under the `<system.web>` element.

Code Snippet 4 shows using the `<globalization>` element.

Code Snippet 4:

```
<system.web>
  <globalization culture="auto" uiCulture="auto"
    enableClientBasedCulture="true"/>
  . . .
  . . .
<system.web>
```

In this code, the true value of the `enableClientBasedCulture` attribute instructs ASP.NET to set the `UICulture` and `Culture` properties for a Web page, based on the preferred languages set in the browser sending the request.

12.2 Implementing Localization

You need to implement localization in your application to display content in languages that the user prefers. For that, you need to separate the language-related and the culture-related content from the application code. The two approaches to implement localization are as follows:

- ➔ Using Resource files
- ➔ Using Separate views

12.2.1 Resource Files

When you use resource files, you must first create a base resource file for the application, for example `MyResources.resx`. Next, you must create separate resource files for each culture that your application supports. These resource files will have the name, as shown in the following syntax:

Syntax:

`<base_resource-filename>. <language_code>-<country_code>.resx`

where,

`base_resource-filename`: Is the name of the base resource file, such as `MyResources`.

`language_code`: Is the language code of the resource file, such as `fr` for the French language.

`country_code`: Is the country code of the resource file, such as `FR` for France.

For example, for the fr-FR culture, you need to name the resource file as MyResources.fr-FR.resx.

In Visual Studio.2013, you can create resource files by performing the following tasks:

1. Create a **Resources** folder in your application.
2. Right-click the **Resources** folder and select **Add→New Item**.
3. Select **General** from the **Installed** templates and select **Resources File**.
4. Replace the name given in the **Name** text box with MyResources.resx.

Figure 12.6 shows specifying the name for the resource file.

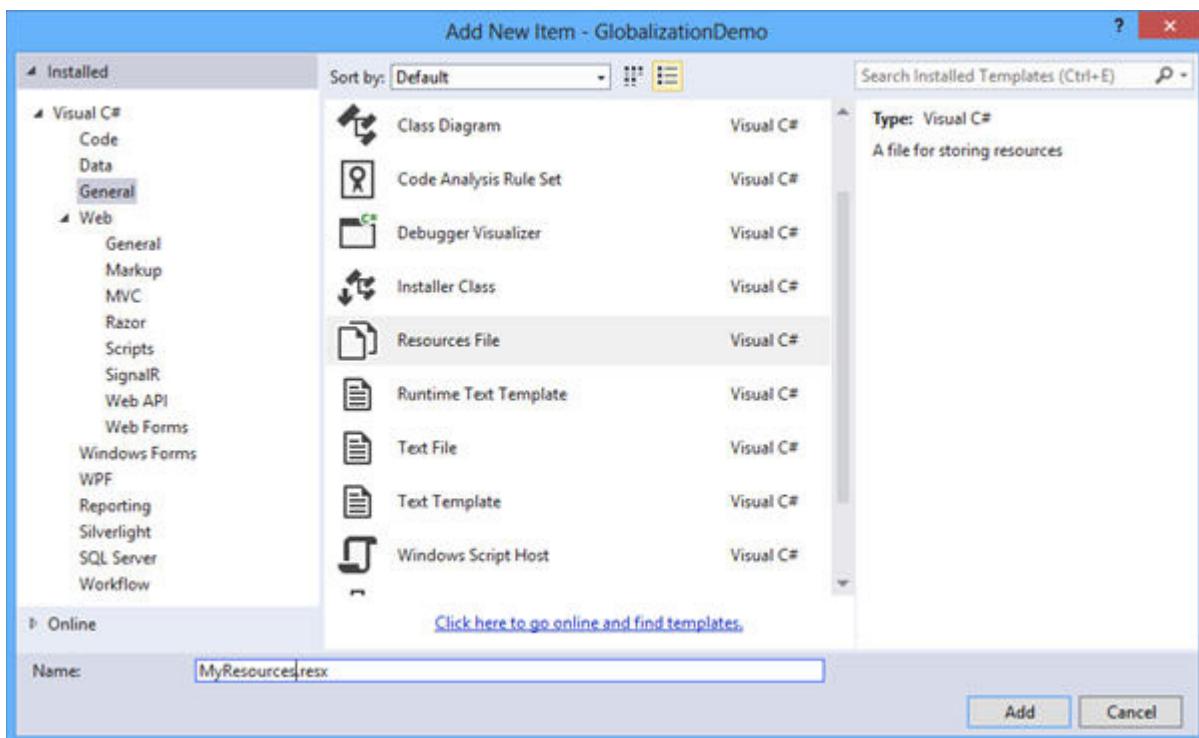


Figure 12.6: Selecting Resources File

5. Click **Add**. The **Resource Editor** displays the MyResources.resx file.
6. Select **Public** from the **Access Modifier** drop-down list. Figure 12.7 shows selecting **Public** from the **Access Modifier** drop-down list.



Figure 12.7: Selecting Public in the Access Modifier List

7. Select MyResources.resx file in the **Solution Explorer** window. The **Properties** window displays the properties of the MyResources.resx file.
8. Type Resources for the **Custom Tool Namespace** property. Figure 12.8 shows specifying values for the property.

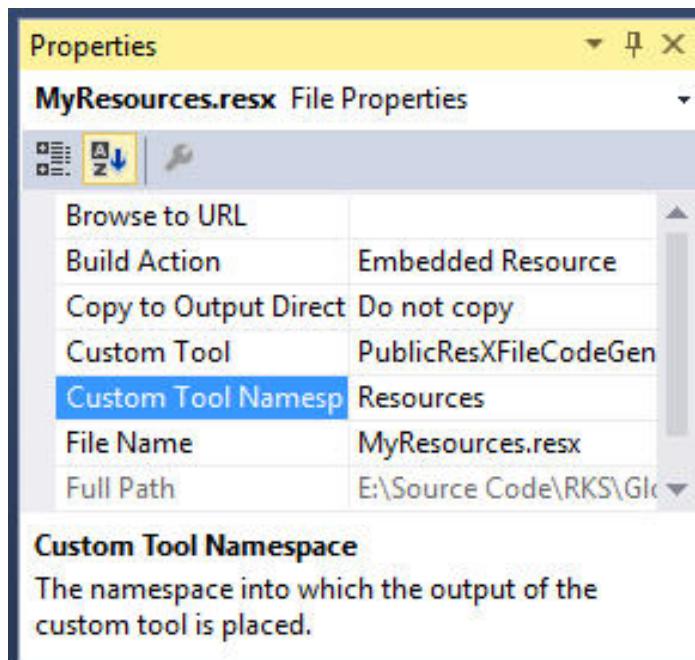


Figure 12.8: Properties Window

9. Add the **Name-Value** pairs in the **Resource Editor**. Figure 12.9 shows specifying the **Name-Value** pairs in the **Resource Editor**.

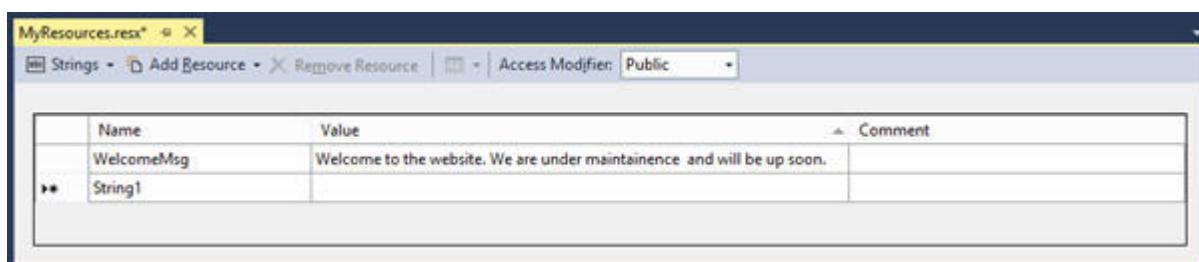


Figure 12.9: Specifying Name-Value Pairs in Resource Editor

Repeat **Step 2** to **Step 7** to create a MyResources.fr-FR.resx file in the **Resources** folder. Then, add the **Name-Value** pairs of the MyResources.fr-FR.resx file in the **Name** and **Value** columns of the **Resource Editor**.

Figure 12.10 shows specifying the **Name-Value** pairs of the `MyResources.fr-FR.resx` file.

Name	Value	Comment
WelcomeMsg	Bienvenue sur le site. Nous sommes en cours de maintenance et sera bientôt.	
*		

Figure 12.10: Name-Value Pairs for `MyResources.fr-FR.resx` File

In the `Index.cshtml` page of the `Home` controller, you can access the value having the `WelcomeMsg` key that you have specified in the resource files.

Code Snippet 5 shows how to access the value having the `WelcomeMsg` key.

Code Snippet 5:

```
<p>@Resources.MyResources>WelcomeMsg</p>
```

A browser whose preferred language is not set to `fr-FR` will render the view using content from the default resource file, named `MyResources.resx` file. Figure 12.11 shows the view using the content from the default resource file.

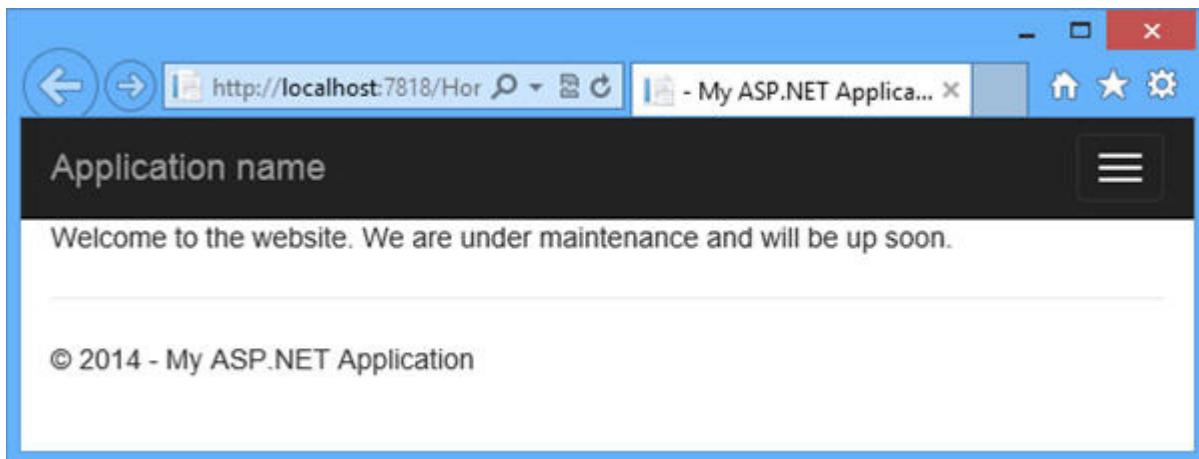


Figure 12.11: Output of Localized View from Default Resource File

A browser whose preferred language is set to `fr-FR` will render the view using content from the resource file, named `MyResources.fr-FR.resx` file.

Figure 12.12 shows the view using content from the `MyResources.fr-FR.resx` file.

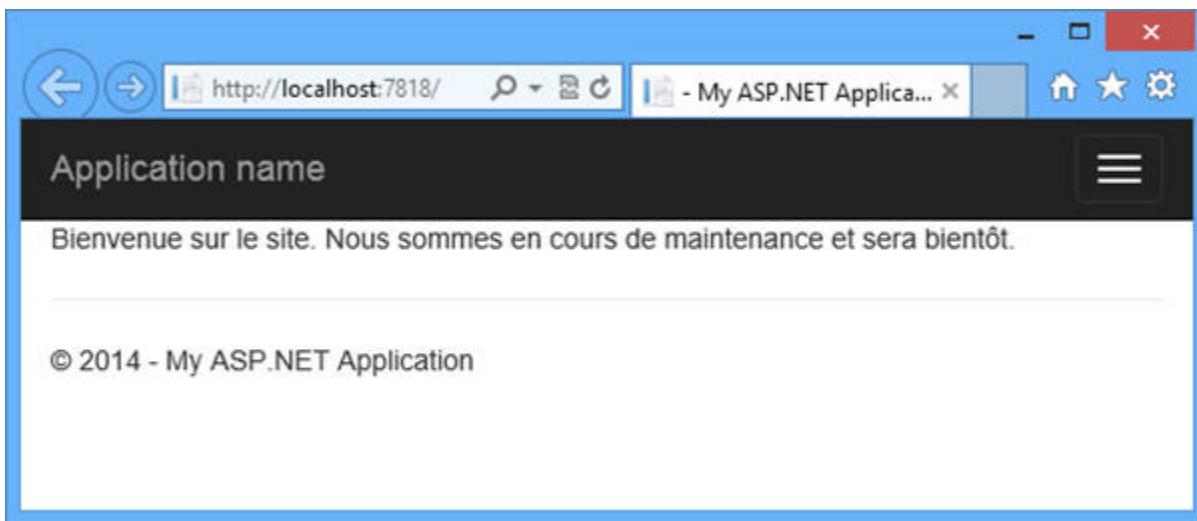


Figure 12.12: Output of Localized View from the `MyResources.fr-FR.resx`

→ Using Separate Views

Consider a scenario where you are using resource files to store the text of labels for the fields of a form. In the view, you have already added the label elements. When you are testing the view for different cultures, you have found instances where localized text that is longer than the length of the label is being displayed partially. In addition, the positioning of the texts with the corresponding form is inconsistent across different cultures.

As a solution, you can use separate views. You can create a view for each of the supported culture and hard code the localized text in them. This will not only result in clean and readable views but will also enable you to control how to position localized text elements in the view.

You can use the approach of using separate views by creating different culture-based views for a particular view that needs to be localized. For example, the `Index.cshtml` view of a Home controller, you can create the following views to support the `en-US` and `fr-FR` cultures:

- `Index.cshtml`: The default view that will be rendered when no specific culture is set.
- `Index.en-US.cshtml`: The view that will be rendered for the `en-US` culture.
- `Index.fr-FR.cshtml`: The view that will be rendered for the `fr-FR` culture.

After creating the views, you can make conditional checks in the controller action to obtain the browser preferred language and accordingly return the corresponding view.

Code Snippet 6 shows the `Index()` action method of a Home controller, that returns different views based on the preferred language of the browser.

Code Snippet 6:

```
public ActionResult Index()
{
    if (HttpContext.Request.UserLanguages[0] == "fr-FR")
        return View("Index.fr-FR");
    else if (HttpContext.Request.UserLanguages[0] == "en-US")
        return View("Index.en-US");
    else
        return View();
}
```

This code uses an `if-else` statement to check the browser preferred language and accordingly returns a corresponding view. If the application does not have a view for a specific language, the action returns the default view.

12.3 Check Your Progress

1. Which of the following options will you use to access all the supported culture of the .NET Framework?

(A)	CultureInfo class
(B)	Page class
(C)	CultureInfo.GetCultures() method
(D)	UICulture property
(E)	UICulture property

(A)	A	(C)	D
(B)	E	(D)	B

2. Which of the following options will you use to explicitly set a specific culture, such as fr-FR or es-ES as a value?

(A)	Culture property
(B)	CultureInfo.GetCultures() method
(C)	UICulture property
(D)	CurrentUICulture property
(E)	SignData() method

(A)	B	(C)	D
(B)	C	(D)	A

3. Which of the following methods of `CultureInfo` class will you use to access the list of supported cultures filtered by the specified type?

(A)	ExportParameters()
(B)	GetCultureInfo()
(C)	GetCulture()
(D)	SignData()
(E)	CurrentCulture()

(A)	B	(C)	A
(B)	C	(D)	E

4. Which of the following term will you use to describe the process of creating an application that is a culture-neutral and language-neutral?

(A)	Internationalization
(B)	Localizability
(C)	Globalizability
(D)	Localization
(E)	Globalization

(A)	B	(C)	D
(B)	E	(D)	A

5. Which of the following code correctly configures the Web.config file to use the browser preferred language?

(A)	<pre><configuration> <globalization culture="auto" uiCulture="auto" enableClientBasedCulture="true"/> </configuration></pre>
(A)	<pre><system.web> <globalization culture="auto" GetCulture="auto" enableClientBasedCulture="true"/> </system.web></pre>
(B)	<pre><system.web> <globalization culture="true" uiCulture="true" enableClientBasedCulture="true"/> </system.web></pre>
(C)	<pre><system.web> <globalization culture="auto" uiCulture="auto" enableClientBasedCulture="true"/> </system.web></pre>
(D)	<pre><system.web> <globalization culture="auto" uiCulture="auto" getClientBasedCulture="false"/> </system.web></pre>

(A)	B	(C)	D
(B)	C	(D)	E

12.3.1 Answers

(1)	A
(2)	B
(3)	B
(4)	B
(5)	C



Summary

- Globalization is a process of designing and developing applications that can be used for multiple cultures.
- In an ASP.NET MVC application, you can access all the supported culture of the .NET Framework using the CultureInfo class.
- In an ASP.NET MVC application, you can set and access the current culture and UI culture of a page from the request processing thread.
- In an ASP.NET MVC application, you can use the Web.config file to use the language that is set as the default language in the browser that sends requests to the application.
- You can implement localization in your application to display content in languages that a user prefers.
- You can use two approaches to implement localization in your application, such as using resource files and using separate views.
- You can use separate views by creating different culture-based views for a particular view that needs to be localized.

Session - 13

Debugging and Monitoring

Welcome to the Session, **Debugging and Monitoring**.

In an ASP.NET MVC application, you can use debugging technique that allows you to identify and then, resolve errors present in the application. Visual Studio 2013 provides various features, such as breakpoint and code stepping that you can use to debug an application.

Once you deploy an ASP.NET MVC application in a Web server, you need to monitor it periodically, to check whether it is functioning properly. For this, an ASP.NET MVC application supports monitoring features that enables you to check for any problems in your application and then, take appropriate measures.

You can also use tools, such as the Performance Monitor and Event Viewer of Windows, to check the performance of an application.

In this Session, you will learn to:

- ➔ Define and describe how to perform debugging
- ➔ Explain how to perform health monitoring of an application
- ➔ Define and describe how to use the Performance Monitoring Tool

13.1 Debugging

While developing an application, it may contain syntax errors, logical errors, and run-time errors. The syntax errors are resolved when you compile the application in Visual Studio 2013. However, other errors such as logical and run-time errors cannot be identified while compiling the application. To identify such errors, you need to debug the code while it is running.

Debugging is a technique that enables you to resolve such errors present in the application. To perform debugging in your application, you can use various techniques that the Visual Studio 2013 provides.

13.1.1 Visual Studio Debugger

There are various tools available that you can use to debug your application. One such tool is Visual Studio debugger. This debugger is available in Visual Studio 2013 IDE. The Visual Studio debugger enables you to run your code line by line, so that you can monitor the program execution properly.

You can use the Visual Studio debugger to check the state of the application objects such as variables and database table values to ensure that the application is running properly. A Visual Studio debugger includes various features that enable you to debug your application.

The session discusses about the features of the Visual Studio debugger that help you to debug ASP.NET MVC applications:

13.1.2 Breakpoints

Breakpoints are places that you can specify in the code where the debugger stops the execution of the application. This allows you to view the current state of data in your application. You can step through each line of code when executing an application.

To set up a breakpoint in Visual Studio 2013, you need to perform the following steps:

1. Open **Visual Studio 2013**.
2. Open a project, for example, the **MVCDemo** project.
3. Click the **AccountController** controller class in the **Solution Explorer** window. The **Code Editor** of Visual Studio 2013 displays the code of the controller class.
4. Locate the `Register()` action method of the controller class.
5. Right-click the beginning of the `if-else` statement of the `Register()` action method and select **Breakpoint→Insert Breakpoint** from the context menu that appears. A red circle on the left of the `if-else` statement of the `Register()` action method is displayed.

Figure 13.1 shows adding a breakpoint.

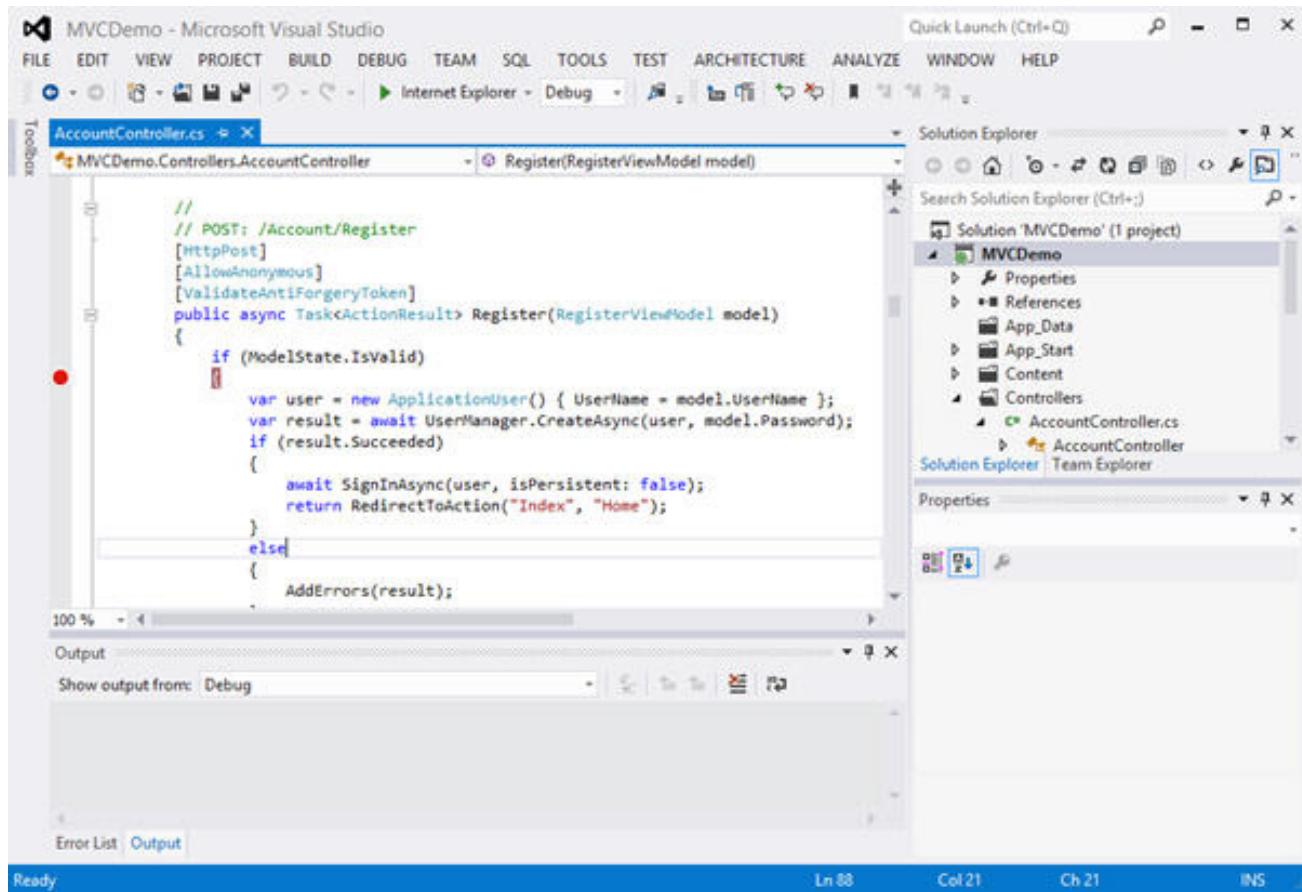


Figure 13.1: Adding Breakpoint

In this figure, the red block over the code line is where the debugger will open.

6. Select **Debug→Start Debugging** from the menu bar of Visual Studio 2013. This will start the debugging process.
7. Click the **Register** link on the **Home** page that appears. The **Registration** page is displayed.
8. In the **Registration** page, type a name in the **User name** text field, type a password in the **Password** field, and type the same password again in the **Confirm password** field.

Figure 13.2 shows specifying data in the **Register** page.

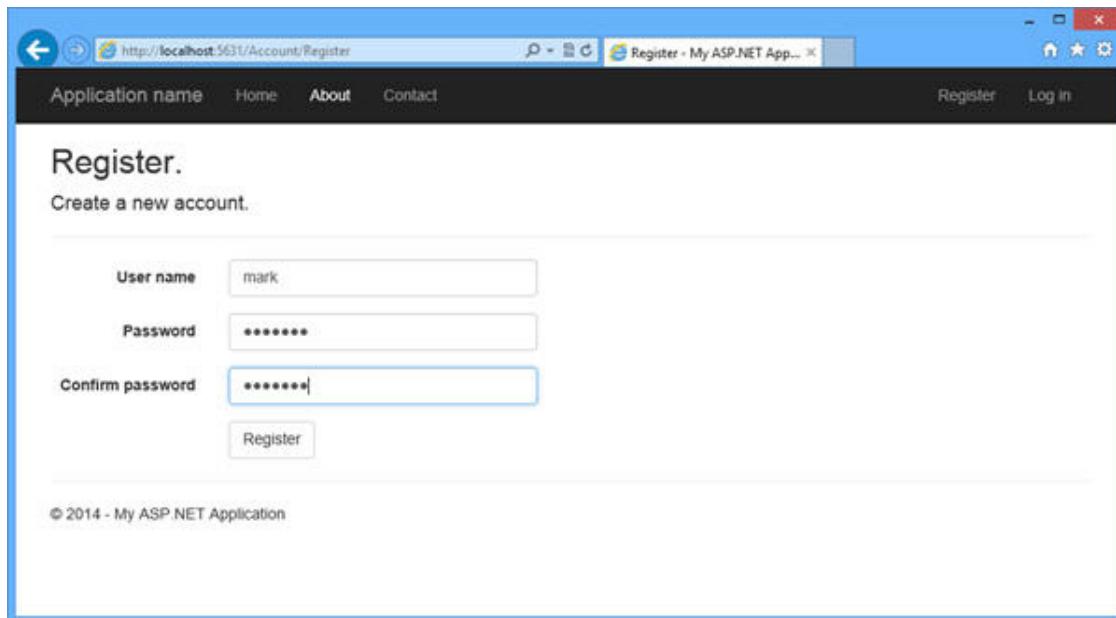


Figure 13.2: Register Page

- Click **Register**. When the program tries to execute the code where the breakpoint is added, the **Code Editor** displays the added breakpoint in yellow color. Figure 13.3 shows the breakpoint when the program tries to execute it.

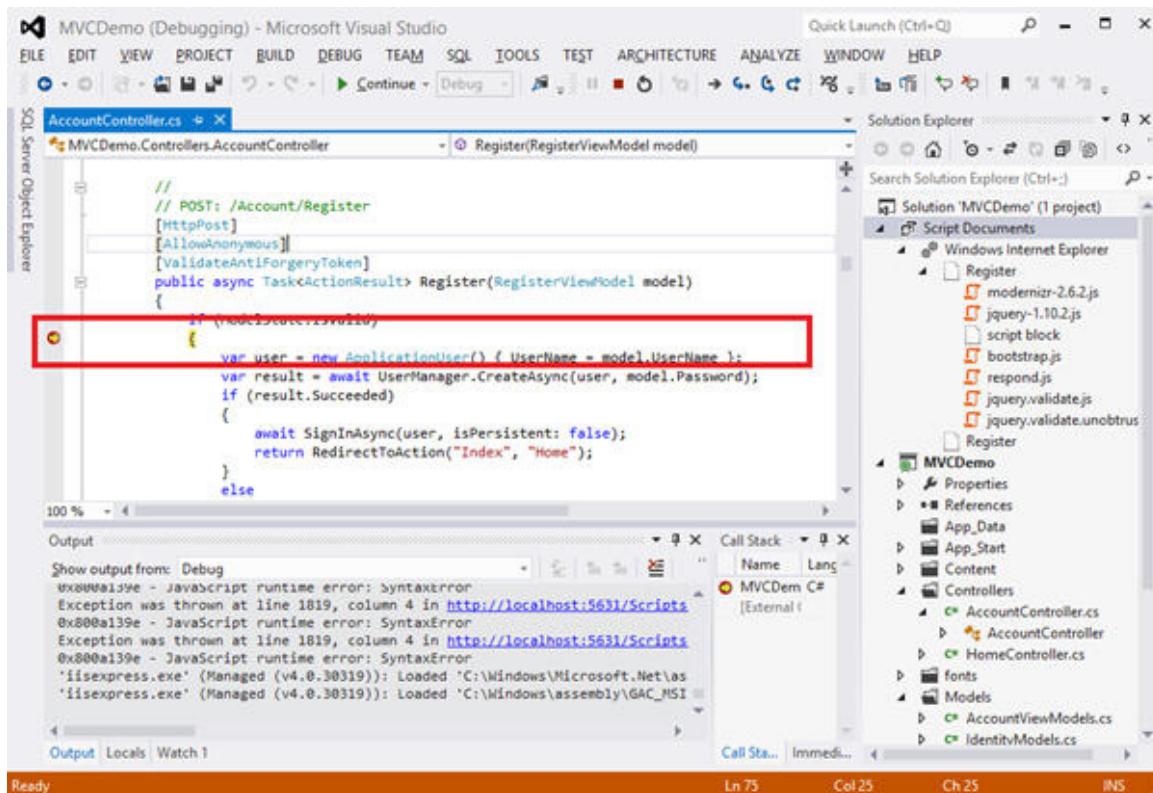


Figure 13.3: Application Paused When Hit the Breakpoint

13.1.3 Viewing Runtime Data

You view runtime data when the application pauses at a breakpoint. By viewing the runtime data, you can analyze whether or not, the application is processing data as expected. To view runtime data, click an object when the application is currently paused at a breakpoint. A drop-down list displays the data of the object. Figure 13.4 shows the data of the `Model` object of the default application.

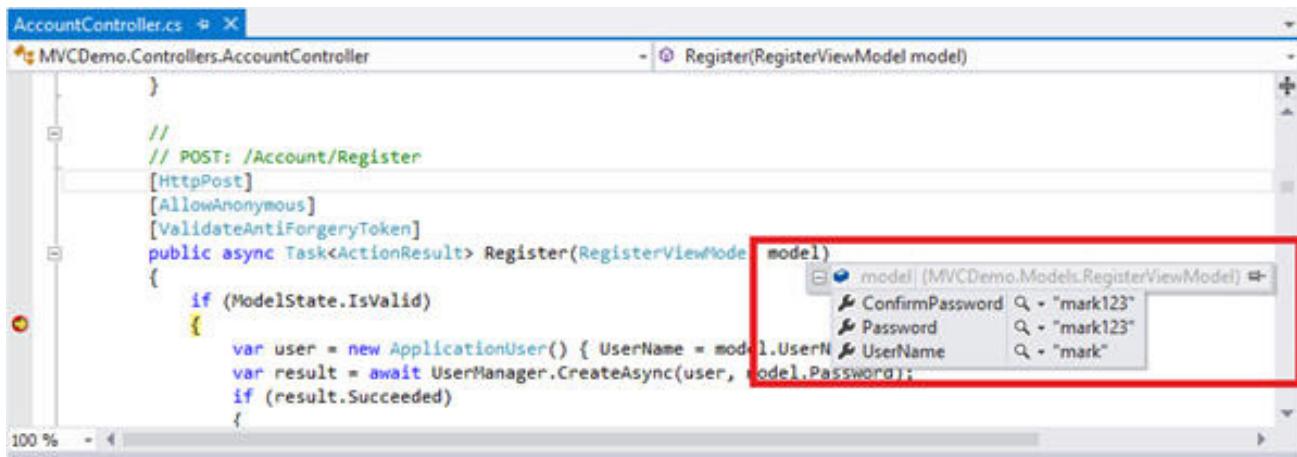


Figure 13.4: Runtime Data

In this figure, the `Model` object stores the data that has been specified in the Registration page of the default application.

13.1.4 Code Stepping

Once you set up breakpoints, you can use certain commands to step through your code line by line. This allows you to carefully examine what each line of code do and how it affects the program. This process of executing the code line by line is known as code stepping.

In Visual Studio 2013, the **Debug** menu contains three steps for debugging procedures, such as Step Into, Step Over, and Step Out.

Both the Step Into and Step Over commands instruct the debugger to execute the next line of code. The only difference between the Step Into and Step Over command is that both of these commands handle function calls differently.

If the code line contains a function call, the Step Into command executes the call itself, and then, paused at the first line of code inside the function. On the other hand, the Step Over commands executes the entire function and then, paused at the first line outside the function.

In short, you can use the Step Into command when you need to look inside a function call, or use the Step Over command when you want to avoid stepping into functions.

You can use the Step Out command when you are inside a function call and want to return to the calling function. The Step Out command resumes execution of your code until the function returns, then breaks at the return point in the calling function.

13.2 Health Monitoring

Once you deploy an ASP.NET MVC application, you need to constantly monitor it for its proper functioning. Sometimes, many unexpected problems, such as Website experiencing heavy load, may occur while the application is running in the real-world environment. By monitoring an application, you can detect the problems occurring in the application and troubleshoot them.

ASP.NET provides health monitoring features that you can use to check for any problems in your application and then, take appropriate troubleshooting technique. Using these features, you can monitor your application while it is running for any issues that could affect its performance.

Monitoring and troubleshooting is required to improve the performance of an application. The ASP.NET Framework provides features, such as health and performance monitoring that you can use to track and monitor an application.

13.2.1 Health Monitoring Features

The health monitoring process allows you to monitor the status of a deployed application. Using this system, you can track system events and errors that affect the application performance.

Health monitoring process allows you to access the detailed run-time information about the resources that an ASP.NET MVC application uses. This information is useful when you need to monitor an application and want to be notified when a critical error occurs in the application. For example, you can use health monitoring system to monitor various events, such as start and end of an application, successful and failed logons, and unhandled exceptions. If you detect any unhandled exception in your application, you can make appropriate changes so that it does not occur again.

In Visual Studio 2013, the **Debug** menu contains a **Start Performance Analysis** command that you can use to check the performance of an application.

To use the **Start Performance Analysis** command in Visual Studio 2013, you need to perform the following steps:

1. Click **Debug→Start Performance Analysis**. The Output window displays the progress of the performance analysis of the application. Figure 13.5 shows the **Output** window.

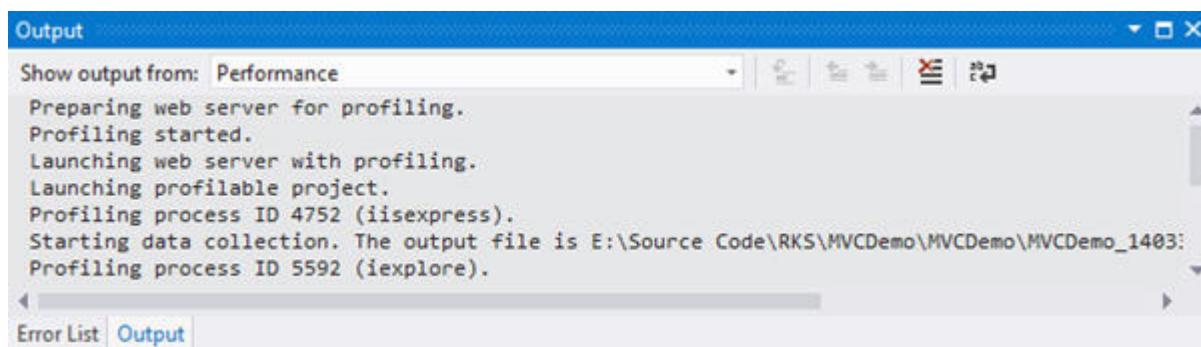


Figure 13.5: Output Window

2. Once the process of performance analysis is started, you can access the application online and start

performing the actions available in the application, such as register to the application and then, logout.

- Close the browser to exit the application. The **Summary** section of Visual Studio 2013 displays the performance of the application in a graphical way. Figure 13.6 shows the performance of an application.

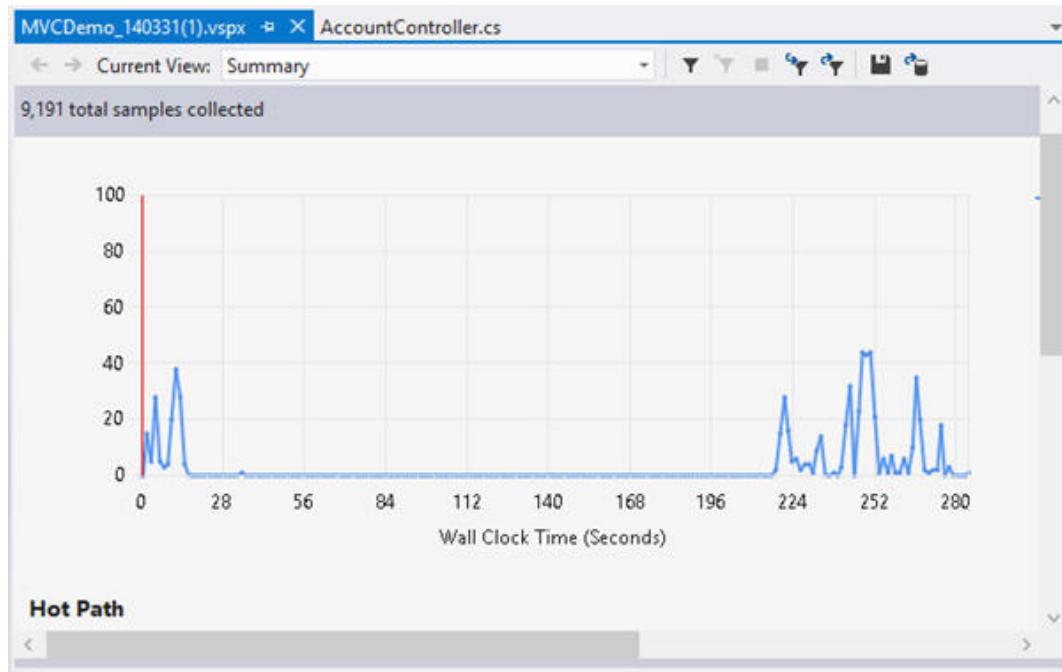


Figure 13.6: Performance of an Application

13.2.2 Performance Monitoring Tools

The Windows Operating System (OS) provides the Performance Monitor tool that you can use to identify any system-level and application-level performance issues. As a developer you can use the data that the Performance Monitor tool provides during the execution of processes as a form of graphical interface. Based on the data, you can identify issues related to your application or related to the existing hardware environment, such as memory, disk, processor, and network.

In the Performance Monitor tool, each of the resources that you monitor is known as Performance Object. Further, each of these objects provide counters representing data on specific aspects of a system or service.

To use the Performance Monitor tool, you need to perform the following tasks:

- Open **Control Panel**.
- Click **Administrative Tools** icon in the **Control Panel** window. The **Administrative Tools** window is displayed.

Figure 13.7 shows the **Administrative Tools** window.

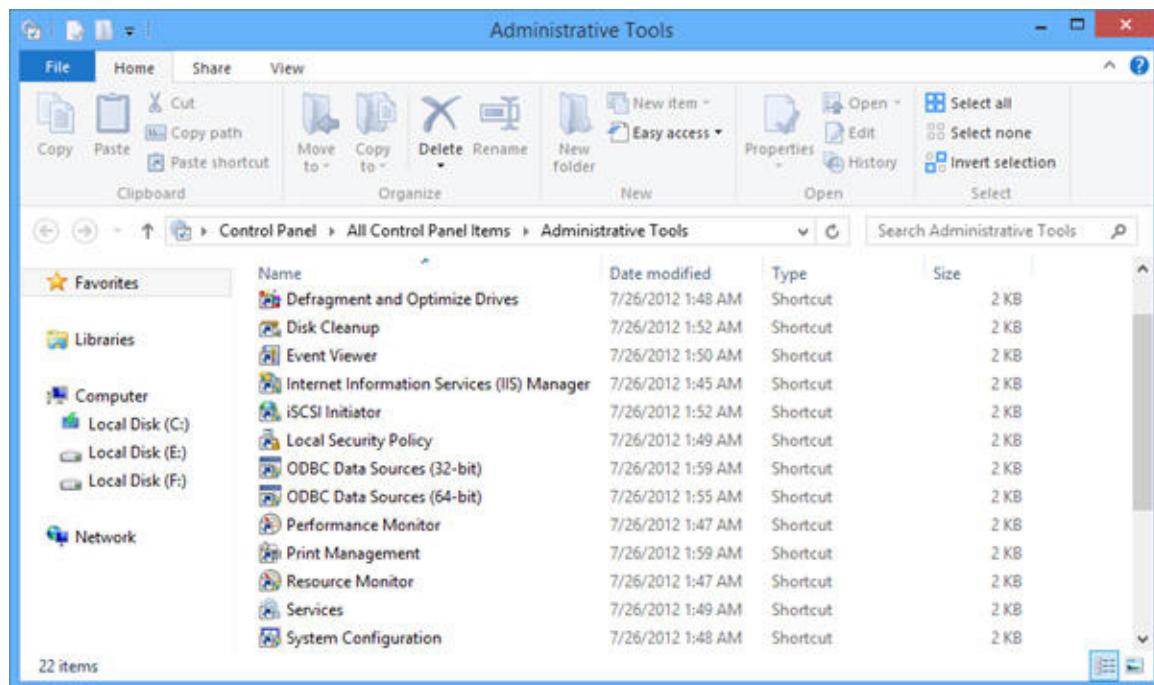


Figure 13.7: Administrative Tools Window

3. Double-click the **Performance Monitor**. The **Performance Monitor** window is displayed. By default, it displays the performance of memory, processors, and physical disks. Figure 13.8 shows the **Performance Monitor** window.

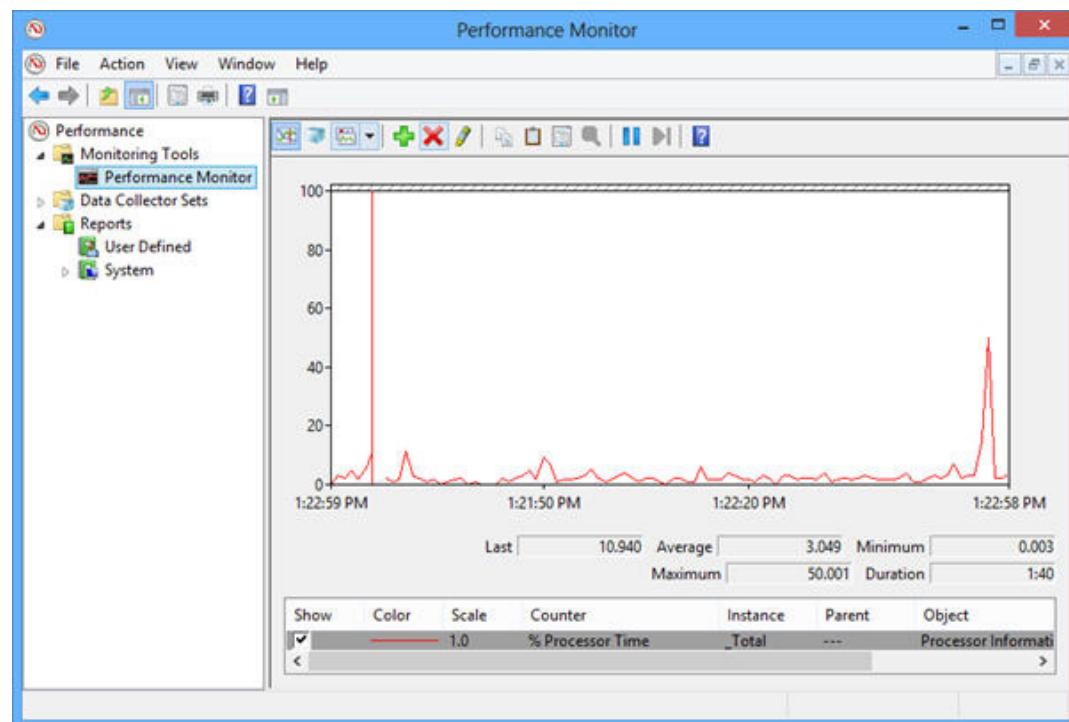


Figure 13.8: Performance Monitor Window

13.2.3 Analyzing Monitoring Results

Once you have viewed the default performance results, you can also analyze the monitoring data of different resources. To analyze the monitoring result of a selected counter, you need to perform the following steps:

1. Right-click the graph and select **Add Counters** from the context menu to add a counter to the monitor. The **Add Counters** dialog box is displayed.
2. Select a counter from the **Select counters from computer** drop-down list. Figure 13.9 shows the **Add Counters** dialog box.

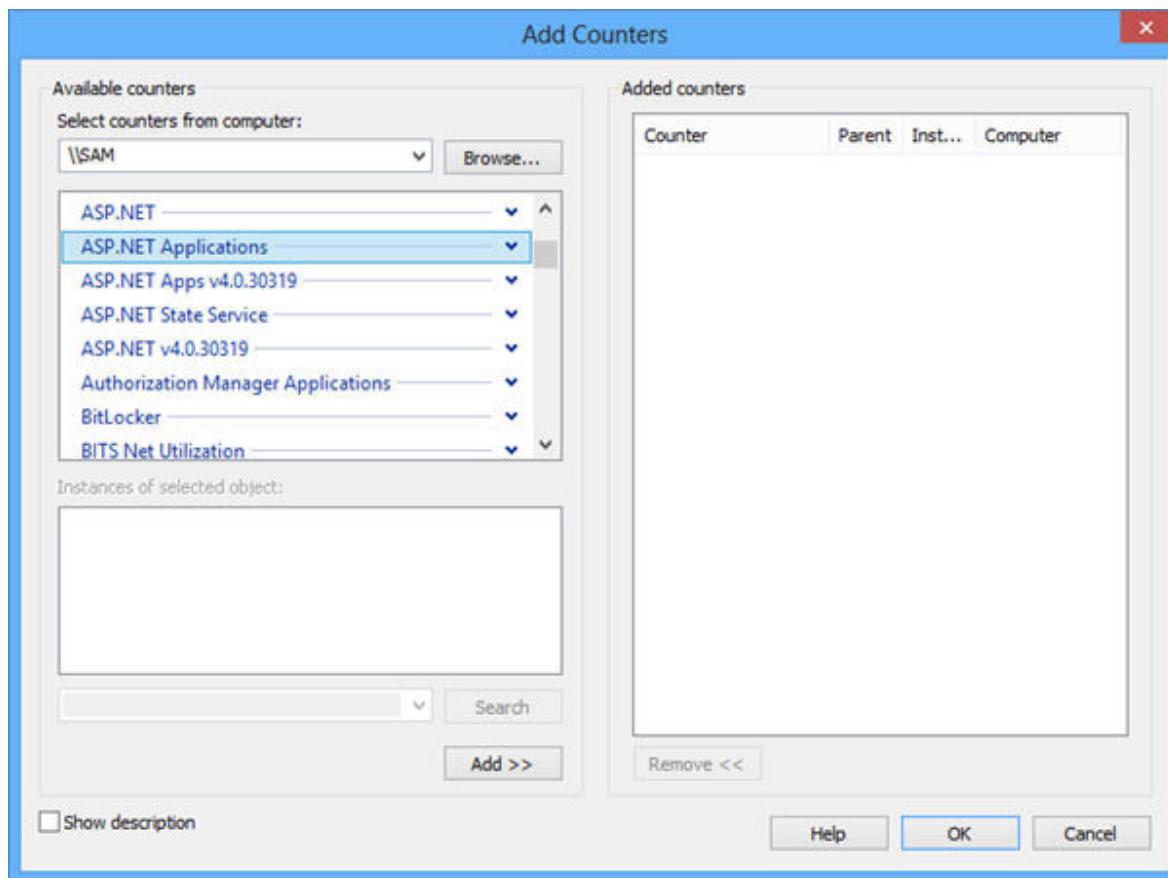


Figure 13.9: Add Counters Dialog Box

3. Select the counters you want to monitor and click **Add**. The **Added counters** section displays the currently added counter.
4. Click **OK**. The **Performance Monitor** window displays the real-time state of the newly added counter graphically.

Figure 13.10 shows the real-time state of the newly added counter.

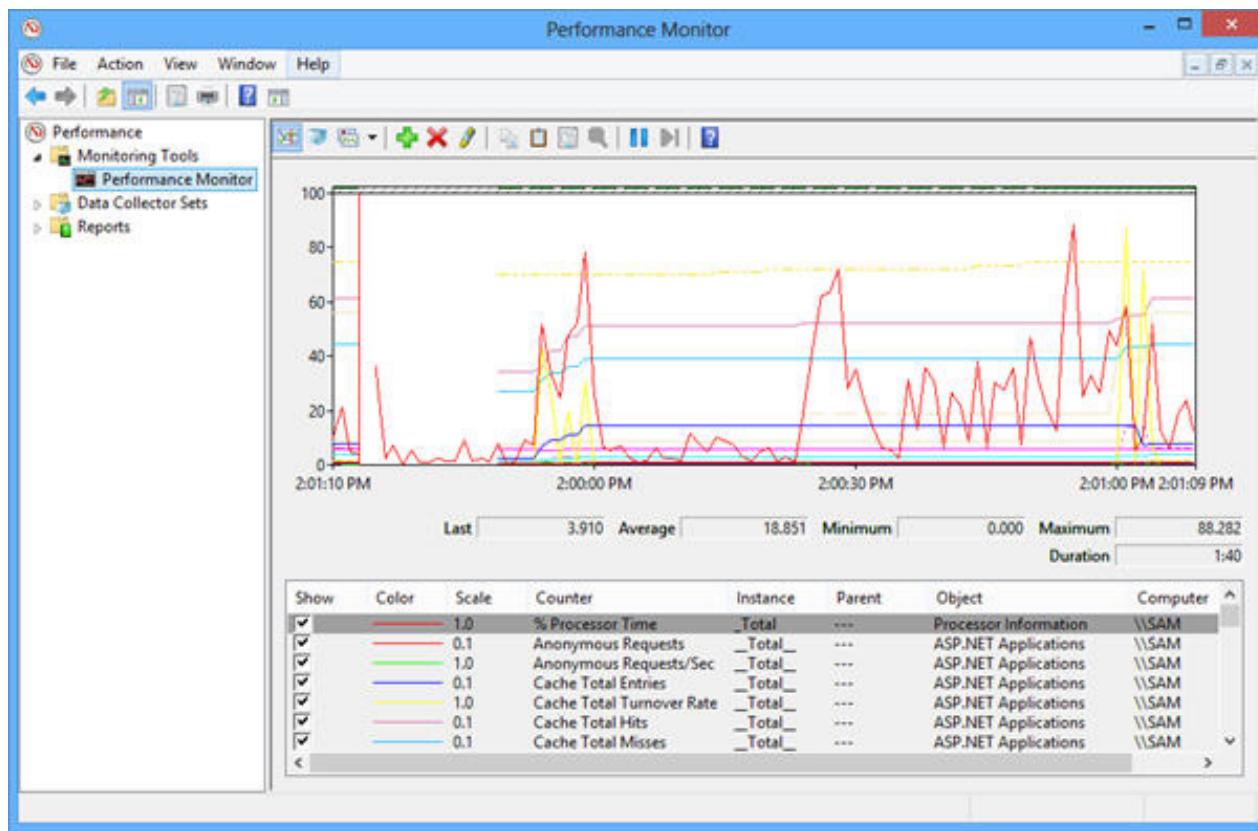


Figure 13.10: Newly Added Counter Performance

13.2.4 Using Event Viewer

The Microsoft Management Console (MMC) provides an interface known as Event Viewer. You can use this interface to view the entries of various event logs. To access the Event Viewer interface to view event logs, you need to perform the following steps:

1. Open **Control Panel**.
2. Click **Administrative Tools** icon in the **Control Panel** window. The **Administrative Tools** window is displayed.
3. Double-click the **Event Viewer**. The Event Viewer window is displayed. By default, it displays the **Application**, **Security**, and **System** logs.

Figure 13.11 shows the **Event Viewer** window.

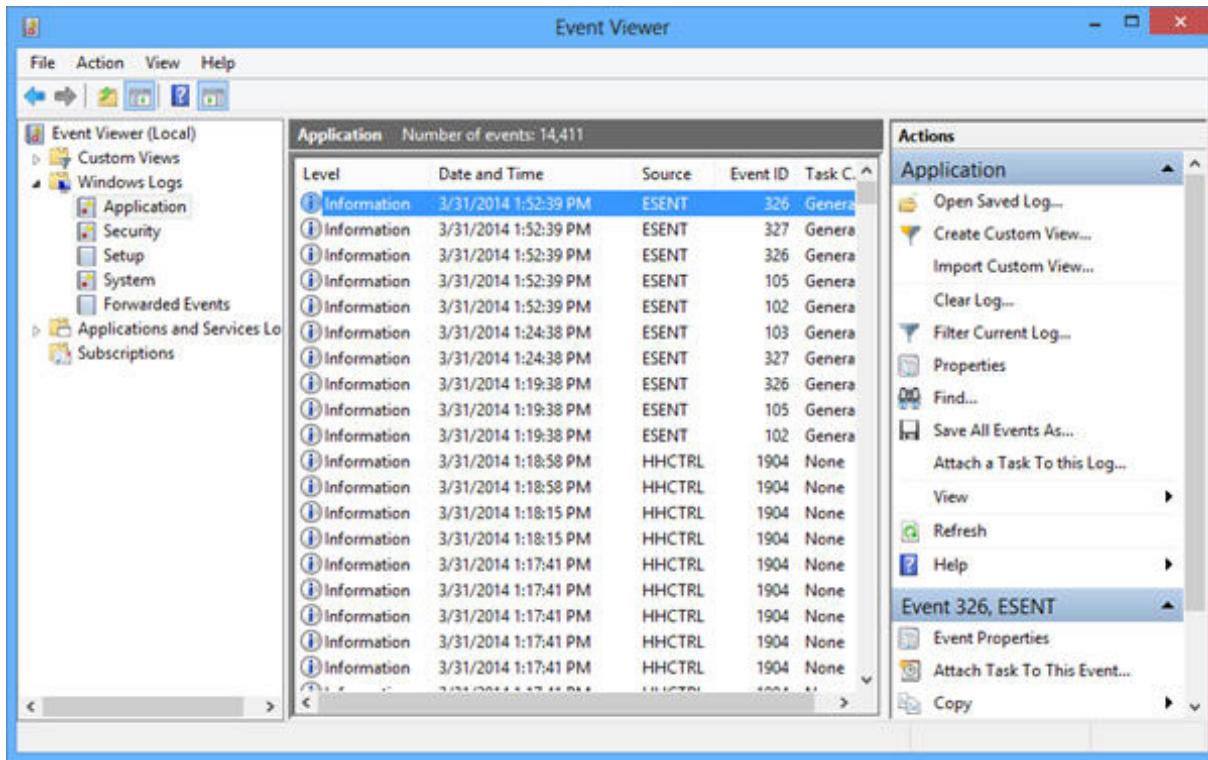


Figure 13.11: Event Viewer Window

4. Click any one of the logs to view its entries.

13.3 Check Your Progress

1. Which of the following options will you use to identify any system-level and application-level performance issues?

(A)	Monitor tool
(B)	Performance Monitor tool
(C)	Health Monitor tool
(D)	Visual Studio debugger
(E)	Breakpoints

(A)	A	(C)	D
(B)	E	(D)	B

2. Which of the following options will you use to view the entries of various event logs?

(A)	Event Viewer
(B)	Code Stepping
(C)	Entry Log Viewer
(D)	Performance Monitor Tool
(E)	Runtime Data View

(A)	B	(C)	D
(B)	C	(D)	A

3. Which of the following options allows you to add a counter to analyze the monitoring result?

(A)	Start Performance Analysis command
(B)	Visual Studio debugger
(C)	Performance Monitor tool
(D)	Localization
(E)	Resource Monitor tool

(A)	B	(C)	A
(B)	C	(D)	E

4. Which of the following commands will you use when you are inside a function call and want to return to the calling function?

(A)	Step Into
(B)	Step Out
(C)	Step Over
(D)	Breakpoint
(E)	Start Performance Analysis

(A)	B	(C)	D
(B)	E	(D)	E

5. Which of the following options allows you to stop the execution of an application and then, view the current state of data in the application?

(A)	Code Stepping
(B)	Step Over command
(C)	Step Out command
(D)	Start Performance Analysis command
(E)	Breakpoints

(A)	B	(C)	D
(B)	C	(D)	E

13.3.1 Answers

(1)	D
(2)	D
(3)	B
(4)	A
(5)	D



Summary

- Debugging is a technique that enables you to resolve such errors present in the application.
- Visual Studio debugger enables you to run your code line by line, so that you can monitor the program execution properly.
- Breakpoints are places that you can specify in the code where the debugger stops the execution of the application and thus, you can view the current state of data in your application.
- You view runtime data when the application pauses at a breakpoint, so that you can analyze whether or not the application is processing data as expected.
- Once you analyze the runtime data, you can use code stepping that allows executing the code line by line.
- The health monitoring process allows you to monitor the status of a deployed application.
- The Windows operating system provides the Performance Monitor tool that you can use to identify any system-level and application-level performance issues.

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION



Session - 14

Advanced Concepts of ASP.NET MVC

Welcome to the Session, **Advanced Concepts of ASP.NET MVC** .

The ASP.NET MVC Framework provides several components to handle client requests. This request handling process involves routing, controller creation, action creation, and view generation.

The ASP.NET MVC Framework also provides filters, which are components that performs some functions before and after an action method executes. These filters can be categorized into authorization, action, result, and exception filters.

Apart from the ASP.NET MVC Framework, you can create Web services for different clients using the Web API Framework.

In this Session, you will learn to:

- ➔ Explain request handling
- ➔ Define and describe filters
- ➔ Explain Web API

14.1 Request Handling

MVC Framework provides several components that together function to handle requests coming from a client. Figure 14.1 shows how MVC Framework performs request handling.

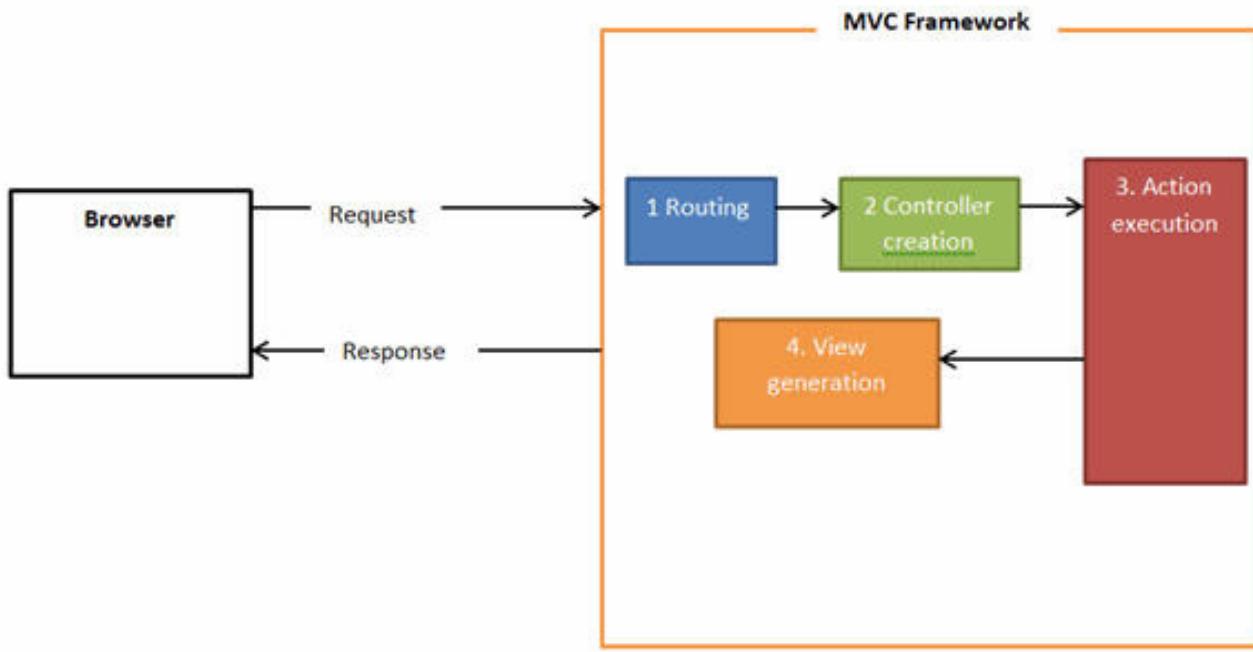


Figure 14.1: Request Handling Process

In the preceding figure, the different steps to handle a request are as follows:

- Routing:** In this step, the Framework maps the URL of the incoming request with URL patterns present in the route table. If a matching pattern is found the request is forwarded to an `IRouteHandler` object. Otherwise, a 404 HTTP status code that indicates non availability of the requested resource is sent back to the client. When a matching pattern is found, the `IRouteHandler` object creates the `MvcHandler` object.
- Controller creation:** In this step, the `MvcHandler` object uses the `IControllerFactory` objects and creates a controller object that implements the `IController` interface.
- Action execution:** In this step, the `CreateActionInvoker()` method of the controller object is called to obtain a `ControllerActionInvoker` object that determines and executes the action. The execution of the action results in an `ActionResult` object that represents the result of the action execution.
- View generation:** In this step, a view engine is created that generates the view based on the `ActionResult` object. The view is then, sent as a response to the client.

14.1.1 Route Handler

In an ASP.NET MVC application, when a request is received, the routing module matches the URL of the incoming request with route constraints defined for the application. Once a URL matches with a route constraint, the routing module returns a handler for this route. This handler is referred to as a route handler for the request and is an implementation of the `IRouteHandler` interface.

The default route handler of MVC Framework is `MvcRouteHandler`. However, at times you might need to create custom route handler to handle requests. For example, consider that an ASP.NET MVC application needs to send XML content from a file as response when a request is received. The following URL pattern is used for handling the requests:

`http://www.<domain_name>/XML/<xml_file_name>`

In the preceding URL pattern:

- `domain_name`: Is the domain name of the ASP.NET application, such as `mvceexample.com`.
- `xml_file_name`: Is the name of the XML file to access, such as `product`.

When the preceding URL arrives, the default route handler of the application will search for a controller, named `XMLController`. As `XML` in the URL pattern represents a folder that contains the XML file and not a controller name, the application will return a `404` HTTP status code to indicate non availability of the requested resource. In such situations, you can create a custom route handler to receive the request, resolve the URL, and use an HTTP handler to process the request.

14.1.2 HTTP Handler

The HTTP handler handles HTTP requests that the route handler receives as a URL. For a request to an XML file, the HTTP handler will be responsible for loading the file and sending the content of the file as a response. To create an HTTP handler to process requests for XML files, you need to create a class that implements the `IHttpHandler` interface and override the `ProcessRequest()` method. This method accepts an `HttpContext` object that represents information about the HTTP request object. Code Snippet 1 implements the `IHttpHandler` interface.

Code Snippet 1:

```
public class CustomHttpHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        RouteValueDictionary values =
            context.Request.RequestContext.RouteData.Values;
        string path = context.Server.MapPath("~/XML/" + values["name"] +
            ".xml");
    }
}
```

```
XDocument xdoc = XDocument.Load(path);
context.Response.Write(xdoc);
context.Response.Flush();
}

public bool IsReusable
{
    get { return true; }
}
```

In this code, the `CustomHttpHandler` class implements the `IHttpHandler` interface that contains the `ProcessRequest()` method. In this method, the `HttpContext.Request.RequestContext.RouteData.Values` property is accessed to retrieve a `RouteValueDictionary` object that represents a collection of route values. Then, the path to the requested URL is created and the `Load()` method of the `XDocument` class is called to load the XML file as an `XDocument` object that represents an XML document. The `HttpContext.Response.Write()` method is called to send the content of the XML document as response.

14.2 Filters

You have already learned that in an ASP.NET MVC application, you can define action method in a controller class, where each action method performs some function when it is invoked through the user interaction. However, there might be situations where you need to implement some functionality before or after the execution of an action method. In such situations, you need to use filters.

While developing an ASP.NET MVC application, you can use filters at the following levels:

- ➔ **An action method:** When you use filters in an action method, the filter will execute only when the associated action method is accessed.
- ➔ **A controller:** When you use filters in controller, the filter will execute for all the actions methods defined in the controller.
- ➔ **Application:** When you use filters in an application, the filter will execute for all the actions methods present in the application.

The ASP.NET MVC Framework supports different types of filters, such as authorization, action, result, and exception filters.

14.2.1 Authorization Filters

Authorization filters execute before an action method is invoked to make security decisions on whether to allow the execution of the action method. For example, the built-in authorize filter when applied to an action checks whether the current user is authorized to execute the action.

In ASP.NET MVC Framework, the `AuthorizeAttribute` class of the `System.Web.Mvc` namespace is an example of authorization filters. This class extends the `FilterAttribute` class and implements the `IAuthorizationFilter` interface. The authorization filter allows you to implement standard authentication and authorization functionality in your application.

For example, consider that only authenticated users are allowed to access the `ViewPremiumProduct()` action method. For that you need to add the `Authorize` attribute on the `ViewPremiumProduct()` method.

Code Snippet 2 shows adding the `Authorize` attribute on the `ViewPremiumProduct()` method.

Code Snippet 2:

```
[Authorize]
public ActionResult ViewPremiumProduct()
{
    return View();
}
```

This code adds the `[Authorize]` attribute on the `ViewPremiumProduct()` action method.

Once you have added the `Authorize` attribute on the `ViewPremiumProduct()` action method, the access to the corresponding view is restricted. Therefore, whenever any user tries to access this view, a page to authenticate the user is displayed. You can use the `Web.config` file to specify the page to be displayed for user authentication.

Code Snippet 3 shows how to specify the page to be displayed for user authentication.

Code Snippet 3:

```
<authentication mode="Forms">
    <forms loginUrl="~/Account/Login" timeout="1440" />
</authentication>
```

In this code, the `<forms>` element specifies the login URL for the application. Whenever a user tries to access a restricted resource, the user is redirected to the login URL. The `timeout` attribute of the `<forms>` element specifies the amount of time in minutes, after which the authentication cookie expires. It is set to 1440 minutes. Its default value is 30 minutes.

14.2.2 Action Filters

MVC action filters enable you to execute some business logic before and after an action method executes. When you use an action filter, the filter receives the following notifications from the MVC Framework for each action method that the filter applies to:

- ➔ The Framework is about to execute the action.
- ➔ The Framework has completed executing the action.

While creating an ASP.NET application, you can create your own action filter as per your requirements. For example, consider a scenario of an ASP.NET MVC application in which you want to log the messages to indicate that the execution of the action method is about to start and the execution of the action method has been completed. For this, you can use an action filter.

To use a custom action filter in an application, you first need to create it and then, apply it to one or more target actions.

You can create a custom action filter by implementing the `IActionFilter` interface. This interface contains methods that you need to use while creating an action filter.

Table 14.1 describes the methods of the `IActionFilter` interface.

Methods	Description
<code>void OnActionExecuting (ActionExecutingContext filterContext)</code>	This method is called before an action method is invoked. The <code>ActionExecutingContext</code> object represents the filter context that you can use to access information about the current controller, HTTP context, request context, action result, and route data.
<code>void OnActionExecuted (ActionExecutedContext filterContext)</code>	This method is called after invocation of an action method. The <code>ActionExecutedContext</code> object represents the filter context that you can use to access information about the current controller, HTTP context, request context, action result, and route data.

Table 14.1: `IActionFilter` Interface Methods

You can also extend the `ActionFilterAttribute` class to create a custom filter. This class implements the `IActionFilter` interface to provide a base class for filter attributes. To create a custom action filter that creates log messages, you need to create a class that extends from the `ActionFilterAttribute` class. Then, use the `OnActionExecuting()` method to specify the code to log messages.

Code Snippet 4 shows the `CustomActionFilterAttribute` class.

Code Snippet 4:

```
public class CustomActionFilterAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {

        Debug.WriteLine("Action execution is about to start", "Action Filter Log");
    }
    ...
}
```

This code uses the `OnActionExecuting()` method and logs a message when the execution of the action method is about to start.

Next, in the `OnActionExecuted()` method you can log a message to indicate that the execution of the action method has been completed.

Code Snippet 5 shows the code of the `OnActionExecuted()` method.

Code Snippet 5:

```
public override void OnActionExecuted(ActionExecutedContext filterContext)
{
    Debug.WriteLine("Action execution has completed", "Action Filter Log");
}
```

This code uses the `OnActionExecuted()` method and creates a log message once the execution of the action method has been completed.

Once you have created an action filter, you can apply it to the action method of a controller class. Code Snippet 6 shows how to apply the `CustomActionFilter` filter at the `Index()` action method.

Code Snippet 6:

```
public class HomeController : Controller
{
    [CustomActionFilter]
    public ActionResult Index()
    {
        return View();
    }
}
```

This code applies the `CustomActionFilter` filter to the `Index()` action method of the `Home` controller.

When you debug the application, the `CustomActionFilter` filter executes and logs the messages in the output window of Visual Studio 2013.

Figure 14.2 shows the log messages in the **Output** window.

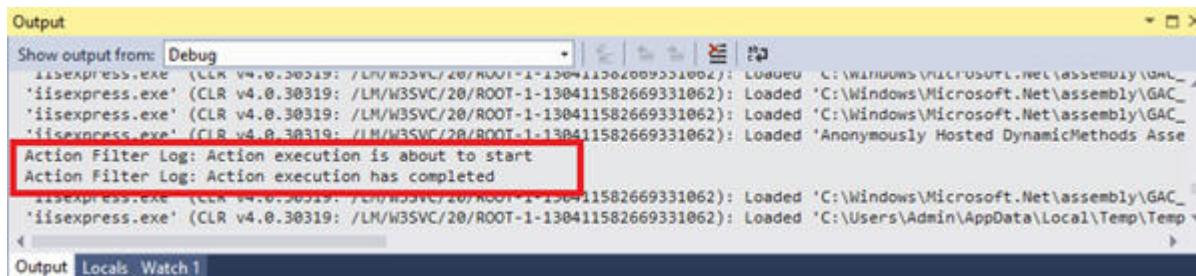


Figure 14.2: Output Window

14.2.3 Result Filters

A result filter operates on the result that an action returns. When you implement a result filter, the filter will receive the following notifications from the MVC Framework for each action method that the filter applies to:

- ➔ The action method is about to execute the action result.
- ➔ The action method has completed executing the action result.

The `OutputCacheAttribute` class is one example of a result filter, which is used to mark an action method whose output will be cached. The `OutputCache` filter indicates the MVC Framework to cache the output from an action method. The same content can be reused to service subsequent requests for the same URL. Caching action output can offer a significant increase in performance, because most of the time-consuming activities required to process a request are avoided.

Code Snippet 7 shows how to use the `OutputCache` attribute.

Code Snippet 7:

```
public class HomeController : Controller
{
    [OutputCache]
    public ActionResult Index()
    {
        // some code
    }
}
```

In this code, the `[OutputCache]` attribute is added to the `Index()` action method of the `HomeController`.

You can specify the duration for which the output of the action should be cached by specifying a `Duration` property with the duration time in seconds.

Code Snippet 8 shows specifying the `Duration` property.

Code Snippet 8:

```
public class HomeController : Controller
{
    [OutputCache(Duration=3)]
    public ActionResult Index()
    {
        //some code
    }
}
```

In this code, the `Duration` property of the `OutputCache` attribute is set to cache the output for 3 seconds.

14.2.4 Exception Filters

In an ASP.NET MVC application, you can use exception filters to handle exceptions that the application throws at runtime. Exception filters are additional exception handling component of MVC Framework besides the built-in .NET Framework exception handling mechanism comprising `try-catch` block.

The MVC Framework provides a built-in exception filter through the `HandleError` filter that the `HandleErrorAttribute` class implements. Like other filters, you can use the `HandleError` filter on an action method or a controller. The `HandleError` filter handles the exceptions that are raised by the controller actions and other filters applied to the action. This filter returns a view named `Error.cshtml` which by default is in the `Shared` folder of the application.

Code Snippet 9 shows the `Error.cshtml` view that displays an error message whenever an action with a `HandleError` filter throws an exception.

Code Snippet 9:

```
@model System.Web.Mvc.HandleErrorInfo
@{
    ViewBag.Title = "Error";
}

<h1 class="text-danger">Error.</h1>
<h2 class="text-danger">An error occurred while processing your request.</h2>
```

This code displays an error message whenever an action with a `HandleError` filter throws an exception.

To use the `HandleError` filter, you need to configure the `Web.config` file by adding a `customErrors` attribute inside the `<system.web>` element. Code Snippet 10 shows how to enable custom errors in the `Web.config` file.

Code Snippet 10:

```
...
<system.web>
...
<customErrors mode="On" />
</system.web>
...
```

In this code, the `On` value of the `mode` attribute in the `<customErrors>` element enables exception handling using the `HandleError` filter.

To test how the `HandleError` filter works, you can update the `Index()` action method of the `Home` controller to throw an exception and handle it using the `HandleError` filter.

Code Snippet 11 shows how to use the `HandleError` filter on the `Index()` action method of the `Home` controller that throws an exception.

Code Snippet 11:

```
public class HomeController : Controller
{
    [HandleError]
    public ActionResult Index()
    {
        throw new DivideByZeroException();
        return View();
    }
    ...
}
```

This code applies the `HandleError` filter to the `Index()` action method of the `Home` controller. The `Index()` action method throws an exception of type `DivideByZeroException`.

When you access the `Index()` action method of the `Home` controller, the `HandleError` filter will catch the exception of type `DivideByZeroException` and display the `Error.cshtml` view.

Figure 14.3 shows the `Error.cshtml` view.

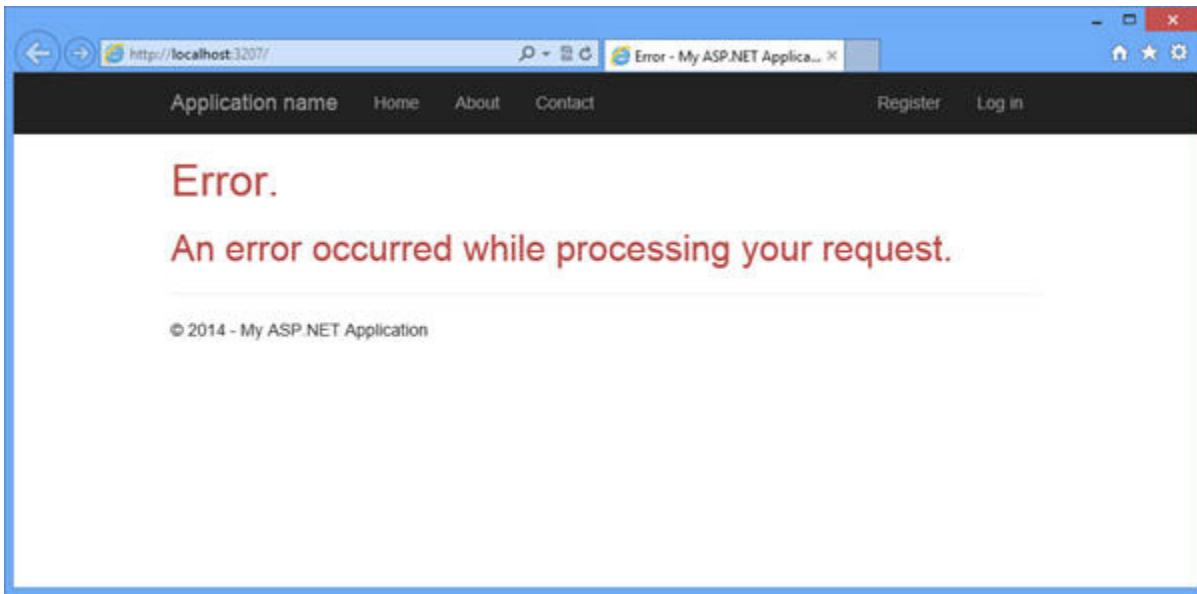


Figure 14.3: Error.cshtml View

14.3 Introduction to Web API

Web API is a Framework that allows you to easily create Web services that provide an API for a wide range of clients, such as browsers and mobile devices using the HTTP protocol.

Using Web API you can display data based on the client requests. For example, if a user is requesting for data to be returned as JSON or XML, Web API handles such requests and returns the data based on the appropriate media type. Web API provides JSON and XML based responses by default.

You can use Web API to develop HTTP based services where client can make a `GET`, `PUT`, `POST`, and `DELETE` requests and access the response provided by Web API.

Web API allows you to add special controllers, known as API Controller. The main characteristics of an API controller are as follows:

- ➔ The action methods return model instead of the `ActionResult`, objects
- ➔ The action methods selected based on the HTTP method used in the request

The model objects that are returned from an API controller action method are encoded as JSON and sent to the client. In addition, as API controllers deliver Web data services, so they do not support views, layouts to generate HTML for browsers to display.

14.3.1 Creating a Web API Application

A Web API application is just a regular MVC Framework application with the addition of a special kind of controller. Visual Studio 2013 allows you to create and build Web API application. To create a new Web API project in Visual Studio 2013, you need to perform the following steps:

1. Open Visual Studio 2013.
2. Click **File→New→Projects** menu options in the menu bar of Visual Studio 2013.
3. In the **New Project** dialog box that appears, select **Web** under the **Installed** section and then, select the **ASP.NET Web Application** template.
4. Type **WebAPIDemo** in the **Name** text field.
5. Click the **Browse** button and specify the location where the application has to be created.
6. Click **OK**. The **New ASP.NET Project – WebAPIDemo** dialog box is displayed.
7. Select **Web API** under the **Select a template** section of the **New ASP.NET Project – WebAPIDemo** dialog box. Figure 14.4 shows selecting **Web API** under the **Select a template** section.

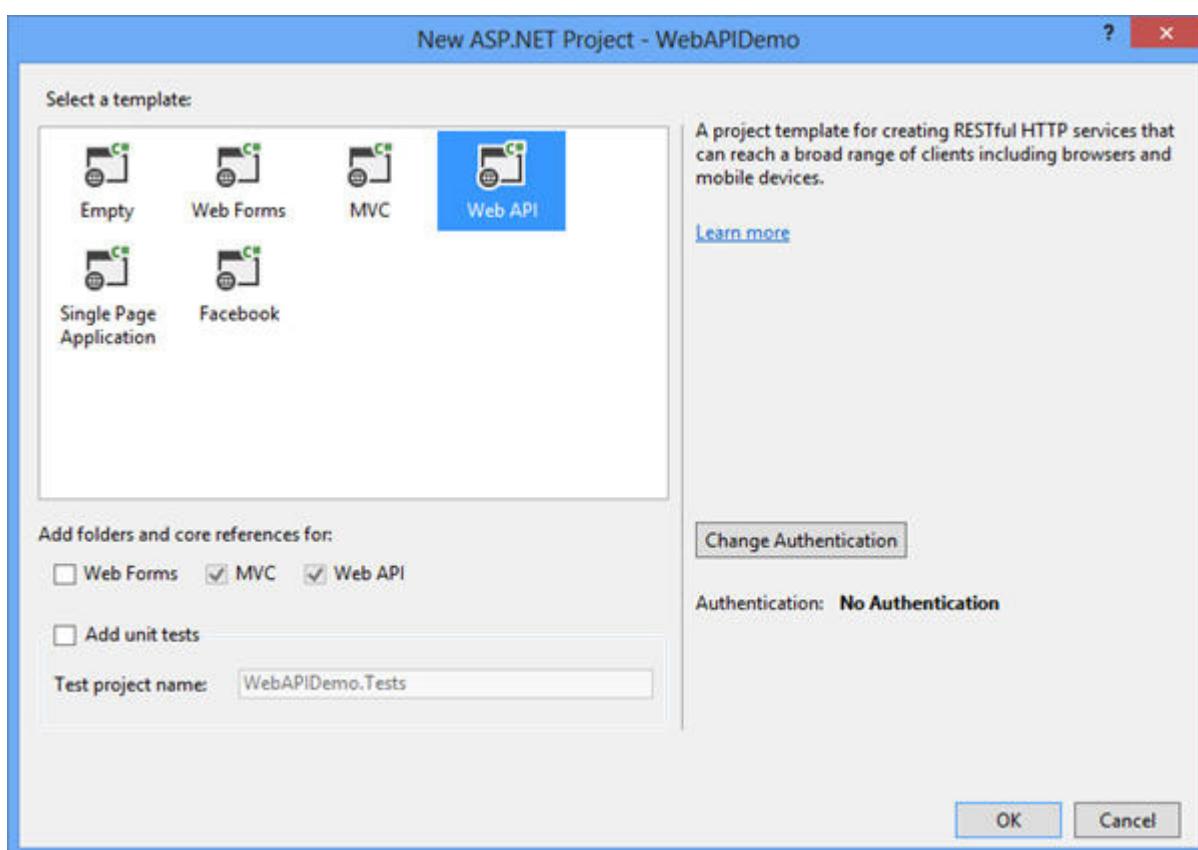


Figure 14.4: Selecting Web API Template

8. Click **OK**. Visual Studio 2013 displays the newly created Web API application. Figure 14.5 shows the newly created Web API application in Visual Studio 2013.

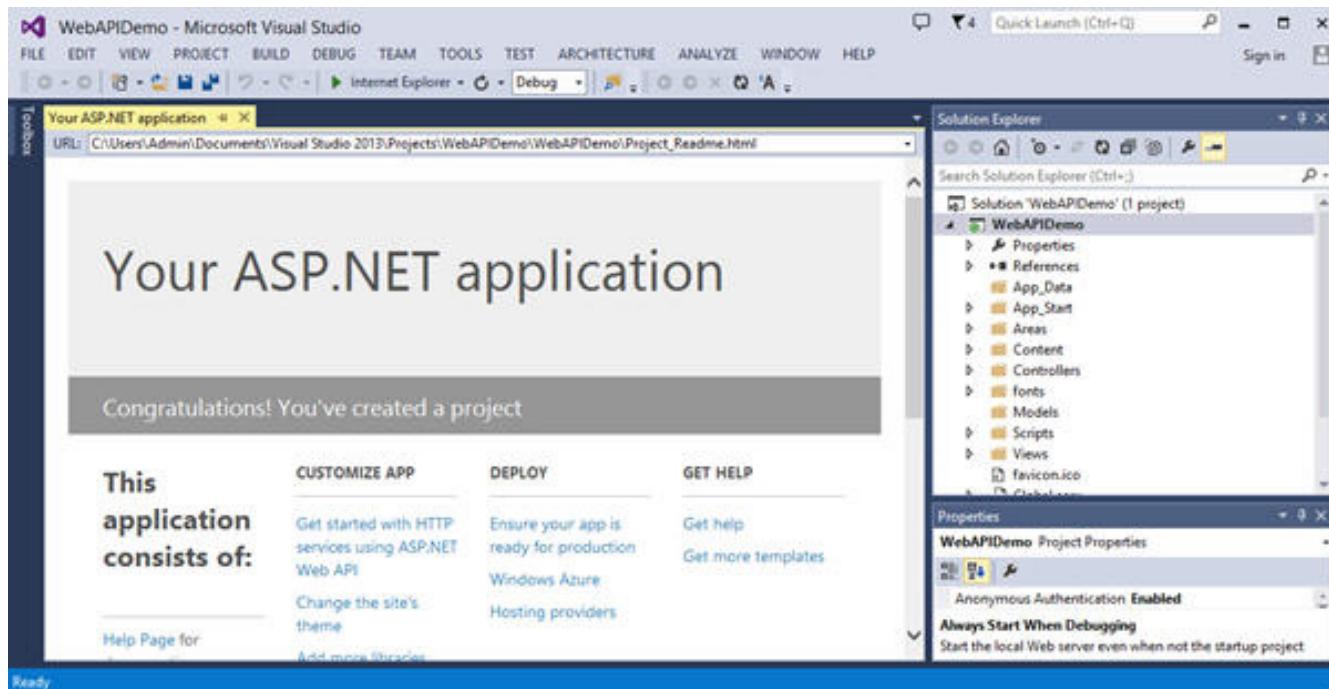


Figure 14.5: Newly Created Web API Application

Note - A Web API application contains all the regular components of an ASP.NET MVC application, such as model objects, controllers and views, and an API Controller.

→ Adding a Model

Once you have created the Web API application, you need to create a model. To create a model in Visual Studio 2013, you need to perform the following steps:

1. Right-click the **Model** folder in the **Solution Explorer** window and select **Add→Class** from the context menu that appears. The **Add New Item – WebAPIDemo** dialog box is displayed.
2. Type **Product.cs** in the **Name** text field of the **Add New Item – WebAPIDemo** dialog box.

Figure 14.6 shows the **Add New Item – WebAPIDemo** dialog box.

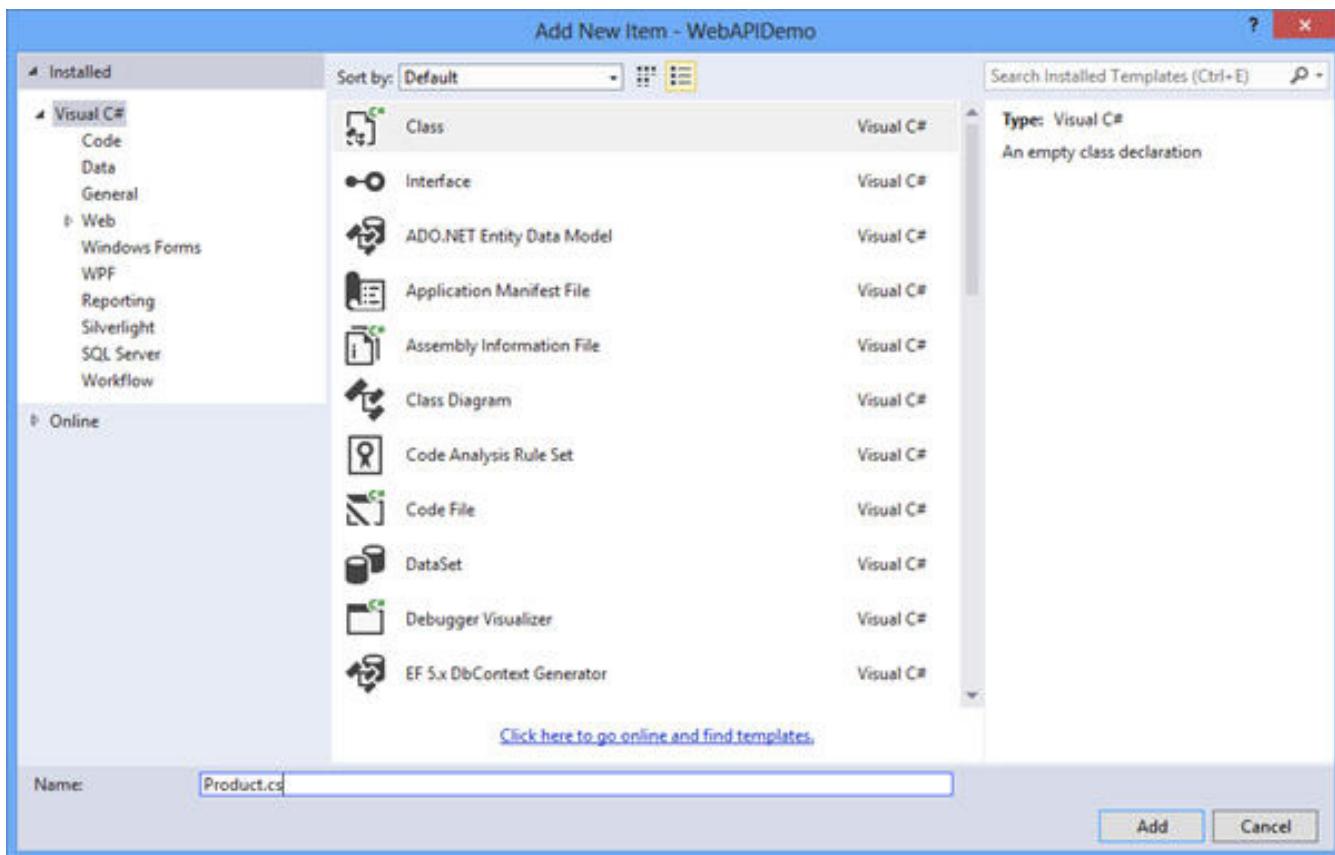


Figure 14.6: Add New Item – WebAPIDemo Dialog Box

3. Click **Add**. The **Code Editor** displays the newly created Product class.
4. In **Code Editor**, add the code to the Product class to represent a product.

Code Snippet 12 shows the Product class.

Code Snippet 12:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
namespace WebAPIDemo.Models
{
    public class Product
    {
```

```

public int Id { get; set; }

public string Category { get; set; }

public string Name { get; set; }

}
}

```

This code declares three variables named `Id`, `Category`, and `Name` along with the `get` and `set` methods.

→ Adding a Web API Controller

Once you have created the model named, `Product.cs` you can add a Web API controller to the application.

In Visual Studio 2013, you can create a controller similar to an ASP.NET MVC application. To create a Web API controller in Visual Studio 2013, you need to perform the following steps:

1. Right-click the **Controllers** folder in the **Solution Explorer** window and select **Add → Controller** from the context menu that appears. The **Add Scaffold** dialog box is displayed.
2. Select the **Web API 2 Controller – Empty** template in the **Add Scaffold** dialog box. Figure 14.7 shows the **Add Scaffold** dialog box.

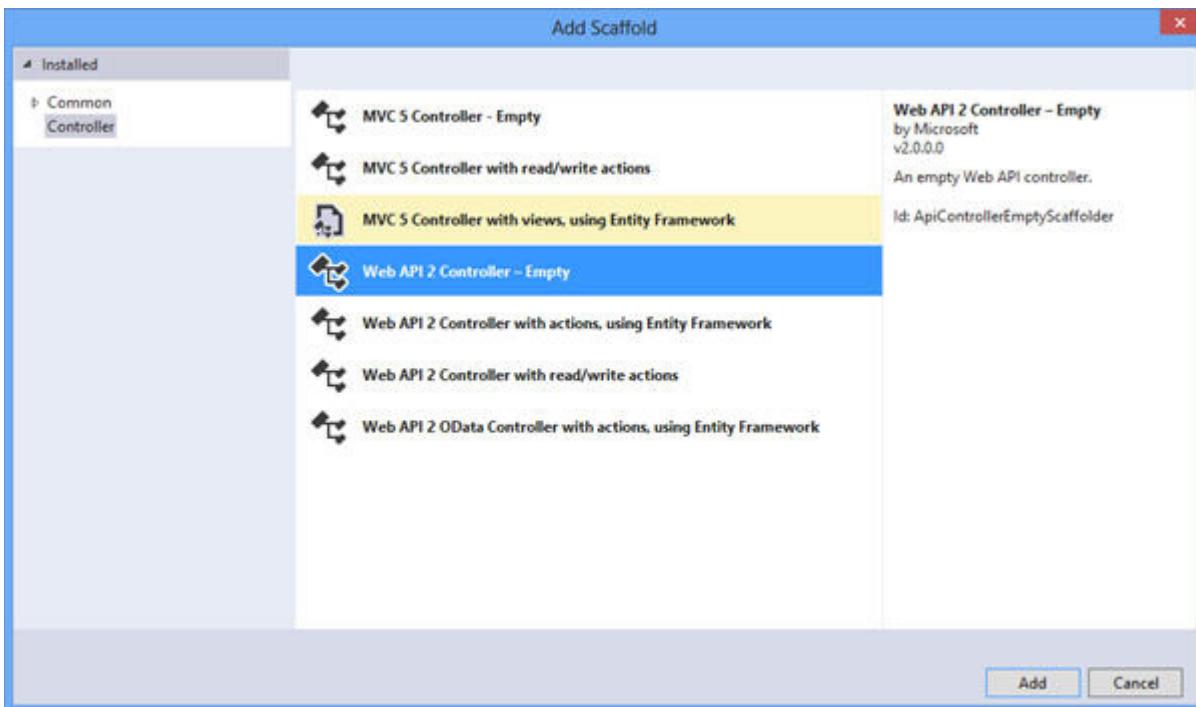


Figure 14.7: Add Scaffold Dialog Box

3. Click **Add**. The **Add Controller** dialog box is displayed.
4. Type `TestController` in the **Controller name** field.

Figure 14.8 shows the **Add Controller** dialog box.

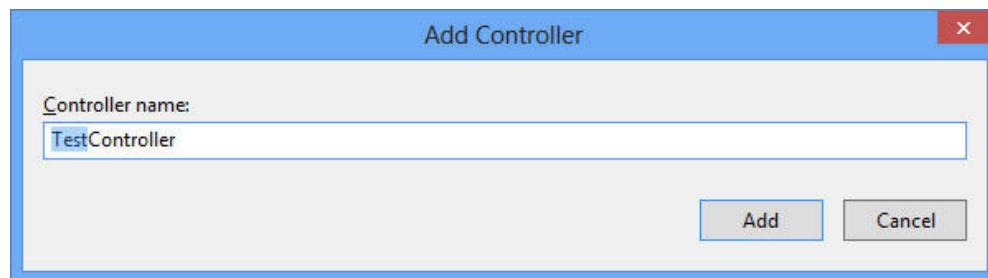


Figure 14.8: Add Controller Dialog Box

5. Click **Add**. The **Code Editor** displays the newly created `TestController` controller class.
6. In the `TestController` controller class, initialize a `Product` array with `Product` objects. Then, create a `Get()` method to return the `Product` array.

Code Snippet 13 shows the `TestController` controller class.

Code Snippet 13:

```
public class TestController : ApiController
{
    Product[] products = new Product[]
    {
        new Product { Id=1, Name = "Product 1", Category= "Category 1" },
        new Product { Id=2, Name = "Product 2", Category= "Category 1" },
        new Product { Id=3, Name = "Product 3", Category= "Category 2" },
    };

    public IEnumerable<Product> Get()
    {
        return products;
    }
}
```

This code snippet defines a `Product` array that contains three `Product` objects. A `Get()` method is defined that will process GET requests that arrives to the controller. The `Get()` method returns the `Product` array.

14.3.2 Defining Routing in Web API

Once you have created the Web API controller, you need to register it with the ASP.NET routing Framework. When the Web API application receives a request, the routing Framework tries to match the URI against one of the route templates defined in the `WebApiConfig.cs` file. If no route matches, the client receives a 404 error.

When you create a Web API application in Visual Studio 2013, by default the IDE configures the route of the application in the `WebApiConfig.cs` file under the `App_Start` folder.

Code Snippet 14 shows the default route configuration of the Web API application.

Code Snippet 14:

```
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

This code shows the default route configuration for a Web API application. This route configuration contains a route template specified by the `routeTemplate` attribute that defines the following pattern:

`"api/{controller}/{id}"`

In the preceding route pattern:

- `api`: is a literal path segment
- `{controller}`: is a placeholder for the name of the controller to access
- `{id}`: is an optional placeholder that the controller method accepts as parameter

You can use the `RouteTable.MapHttpRoute()` method to configure routes of a Web API application. For example, consider the following URL: `http://localhost:9510/web/Test`

Assuming that the preceding URL needs to access the `TestController` controller class of the `WebAPIDemo` application, you need to configure a route in the `WebApiConfig.cs` file.

Code Snippet 15 shows how to configure a route.

Code Snippet 15:

```
config.Routes.MapHttpRoute(
    name: "TestDefaultApi",
    routeTemplate: "web/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

In this code, a route named `TestDefaultApi` is created with the route pattern `web/{controller}/{id}`.

14.3.3 Accessing an ASP.NET Web API Application

Once you have created the controller class and configure the routing of the Web API application, you can access it over the browser. For that, you need to perform the following steps:

1. Click **Debug→Start Without Debugging**. The browser window displays the **Home** page of the application. Figure 14.9 shows the **Home** page of the application.

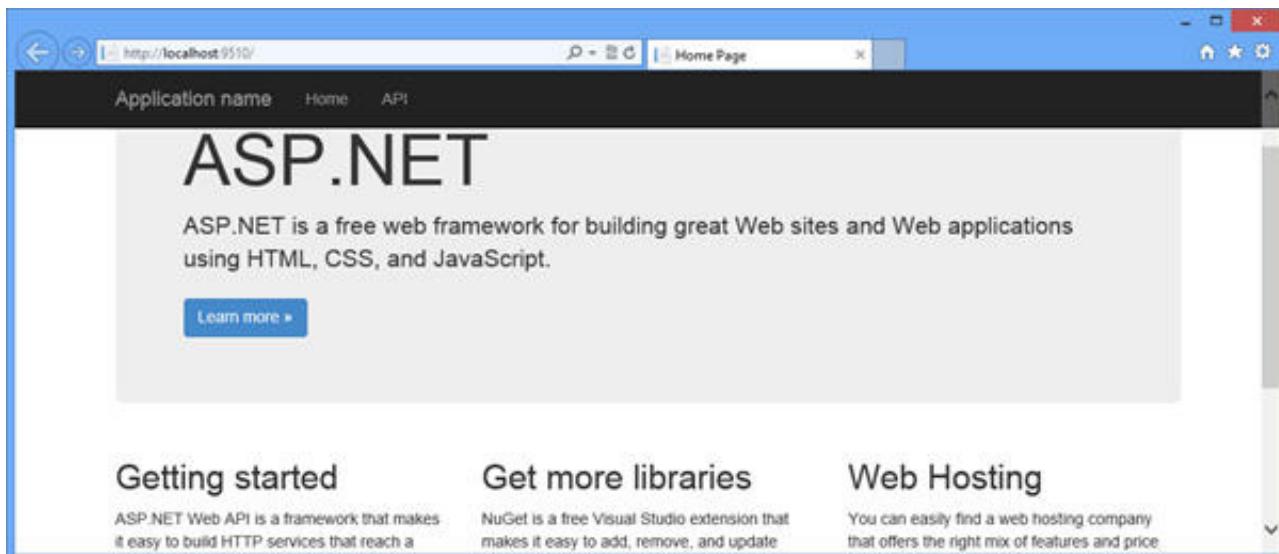


Figure 14.9: Home Page

2. Type the following URL in the address bar of the browser:

`http://localhost:9510/web/Test`

Browser displays the product details.

Figure 14.10 shows the product details.

```
-<ArrayOfProduct>
--<Product>
  <Category>Category 1</Category>
  <Id>1</Id>
  <Name>Product 1</Name>
</Product>
--<Product>
  <Category>Category 1</Category>
  <Id>2</Id>
  <Name>Product 2</Name>
</Product>
--<Product>
  <Category>Category 2</Category>
  <Id>3</Id>
  <Name>Product 3</Name>
</Product>
</ArrayOfProduct>
```

Figure 14.10: Product Details

14.4 Check Your Progress

1. Which of the following classes implements a result filter?

(A)	ActionResult
(B)	ViewResult
(C)	FilterAttribute
(D)	CustomHttpHandler
(E)	OutputCacheAttribute

(A)	A	(C)	D
(B)	C	(D)	E

2. Which of the following method will you use to send the content of an XML document as response?

(A)	HttpContext.Response.Write()
(B)	Load()
(C)	ProcessRequest()
(D)	CreateActionInvoker()
(E)	OnActionExecuting()

(A)	C	(C)	D
(B)	A	(D)	E

3. Which of the following file will you use to define the route templates in an application?

(A)	Global.asax
(B)	Web.config
(C)	WebApiConfig.cs
(D)	RouteConfig.cs
(E)	FilterConfig.cs

(A)	B	(C)	A
(B)	C	(D)	E

4. Which of the following method is called before an action method is invoked?

(A)	OnActionExecuting ()
(B)	OnActionExecuted ()
(C)	OnActionExecutes ()
(D)	ActionExecutingContext()
(E)	ProcessRequest()

(A)	B	(C)	D
(B)	E	(D)	A

5. Which of the following classes will you extend to create a custom action filter?

(A)	FilterAttribute
(B)	AuthorizeAttribute
(C)	ActionAttribute
(D)	ActionFilterAttribute
(E)	ActionFilter

(A)	A	(C)	C
(B)	E	(D)	D

14.4.1 Answers

(1)	D
(2)	B
(3)	B
(4)	D
(5)	D



Summary

- MVC Framework provides several components that together function to handle requests coming from a client.
- In an ASP.NET MVC application, when a request is received, the routing module matches the URL of the incoming request with route constraints defined for the application.
- The HTTP handler handles HTTP requests that the route handler receives as a URL.
- Authorization filters execute before an action method is invoked to make security decisions on whether to allow the execution of the action method.
- MVC action filters enable you to execute some business logic before and after an action method executes.
- Exception filters are additional exception handling component of MVC Framework besides the built-in .NET Framework exception handling mechanism comprising try-catch block.
- Web API is a Framework that allows you to easily create Web services that provide an API for a wide range of clients, such as browsers and mobile devices using the HTTP protocol.

Get
WORD WISE



Visit
Glossary@

www.onlinevarsity.com

Session - 15

Testing and Deploying

Welcome to the Session, **Testing and Deploying**.

Testing is a process of ensuring that an application works properly after deploying it on a Web server. While creating an ASP.NET MVC application in Visual Studio 2013, you create a unit test for the application that enables creating classes and methods in your application and test their intended functionality.

Once you create an ASP.NET MVC application, you should make it available to the users. For this, you need to install your application on a Web server, which is known as deployment. You can use the IIS Web server to develop, host, and manage your application.

In this Session, you will learn to:

- ➔ Define and describe how to perform unit tests
- ➔ Explain how to prepare an application for deployment
- ➔ Define and describe how to deploy an application on IIS

15.1 Testing

Once you have created an application, you should ensure that the application works properly by testing it thoroughly to provide the best quality possible. Sometimes, an application may work properly at the time of development. However, when it is deployed, you cannot be sure of the inputs that the users may provide to the application and the results that the application may generate in such a scenario. To avoid such problems, you should test the application before you deploy it.

15.1.1 Preparing for Unit Tests

Unit testing is a technique that allows you to create classes and methods in your application and test their intended functionality. In the context of unit testing, a unit is the smallest part of an application that can be tested. Unit testing is concerned whether or not the individual units that make up the application functions as expected.

When you create an ASP.NET MVC application in Visual Studio 2013, you can specify whether to create a unit test for the application.

To create an application in Visual Studio 2013 with unit test, you need to perform the following steps:

1. Create a new ASP.NET Web Application project, named UnitTestDemo. Figure 15.1 shows the UnitTestDemo project.

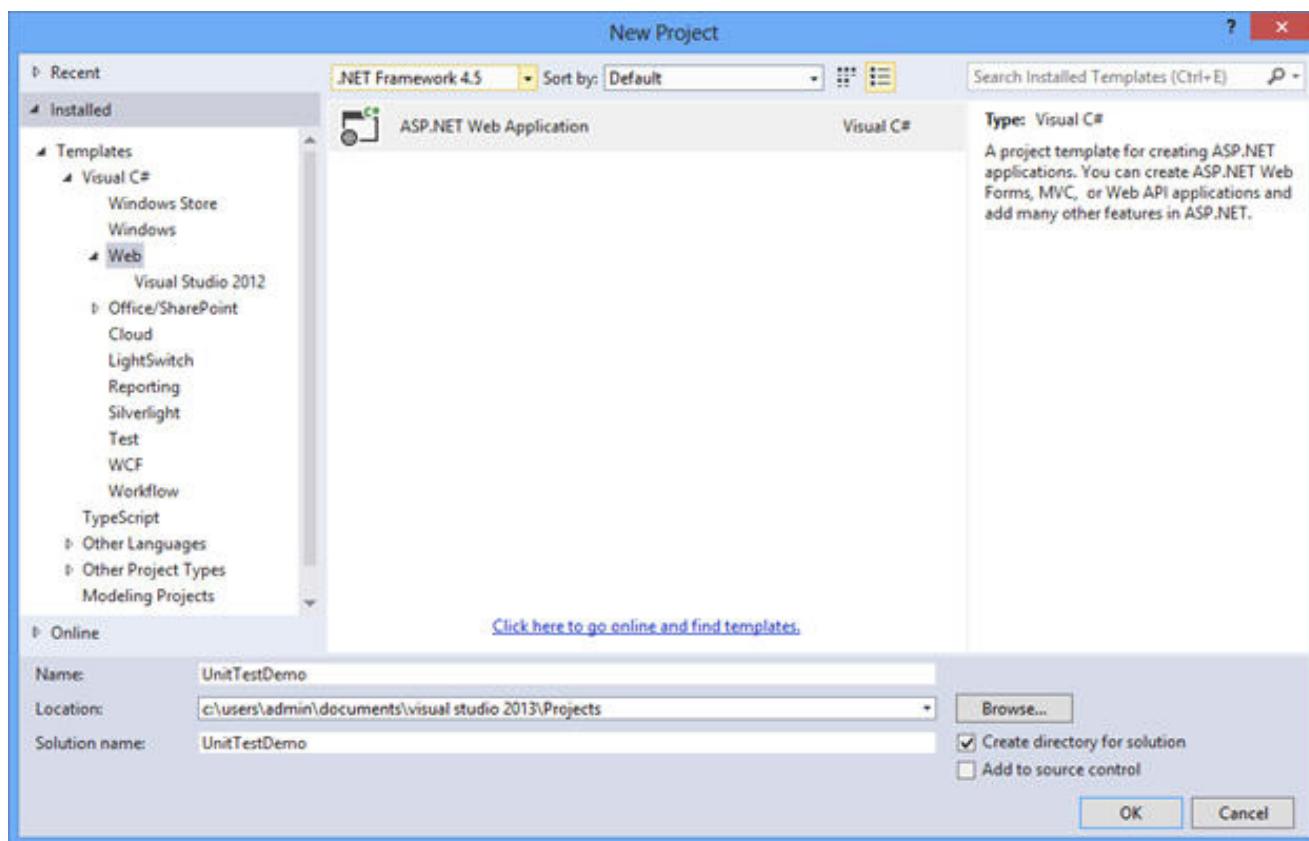


Figure 15.1: UnitTestDemo Project

2. Click **OK**. The **New ASP.NET Project – UnitTestDemo** dialog box appears.
3. Select **MVC** under the **Select a template** section and select the **Add unit tests** check box. Figure 15.2 shows the **New ASP.NET Project – UnitTestDemo** dialog box.

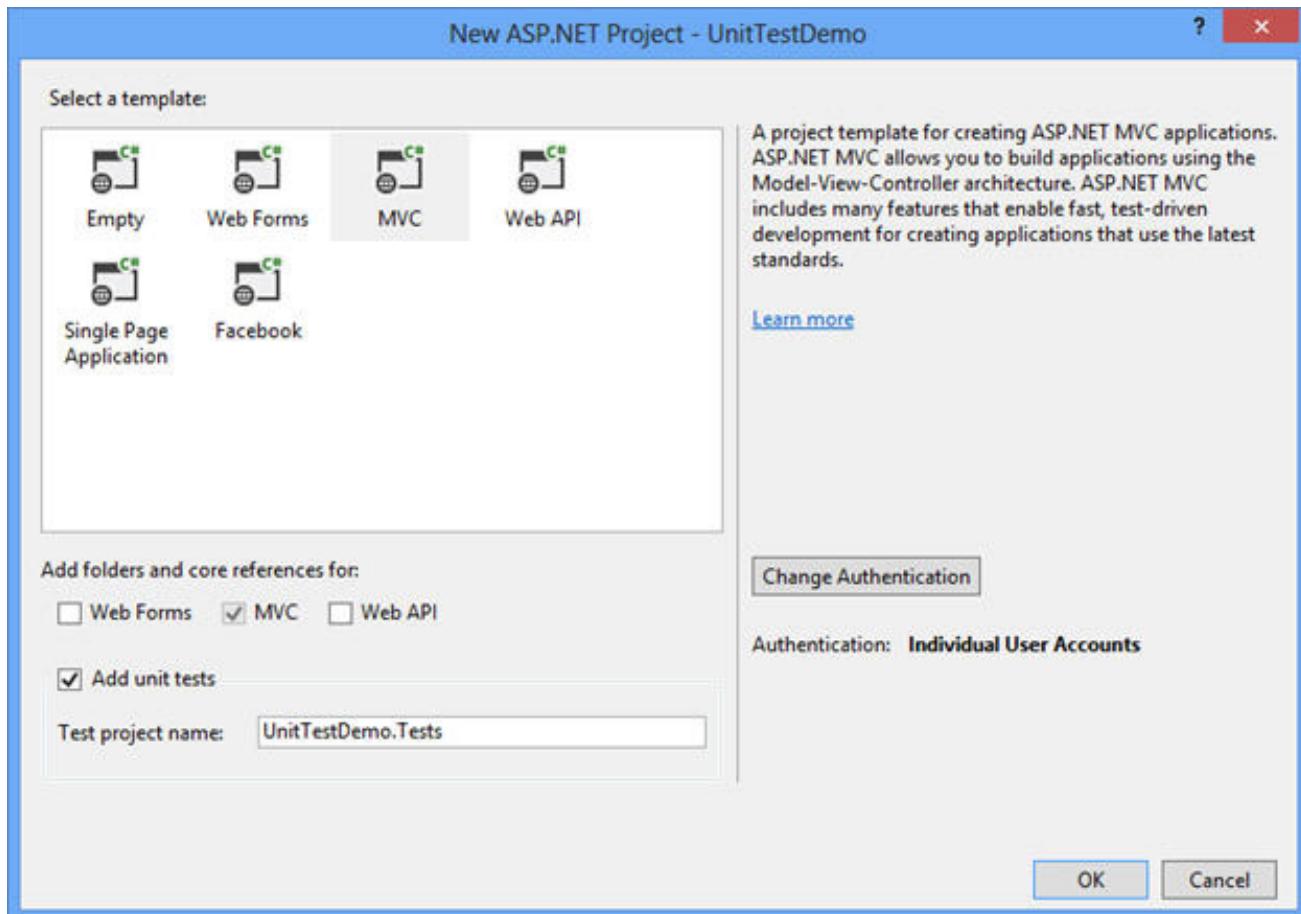


Figure 15.2: New ASP.NET Project – UnitTestDemo Dialog Box

4. Click **OK**. Visual Studio 2013 creates the project with support for unit testing. The **Solution Explorer** window displays a **UnitTestDemo.Tests** node that contains the resources to perform unit test on the application.

Figure 15.3 shows **Solution Explorer** that displays the `UnitTestDemo.Tests` node.

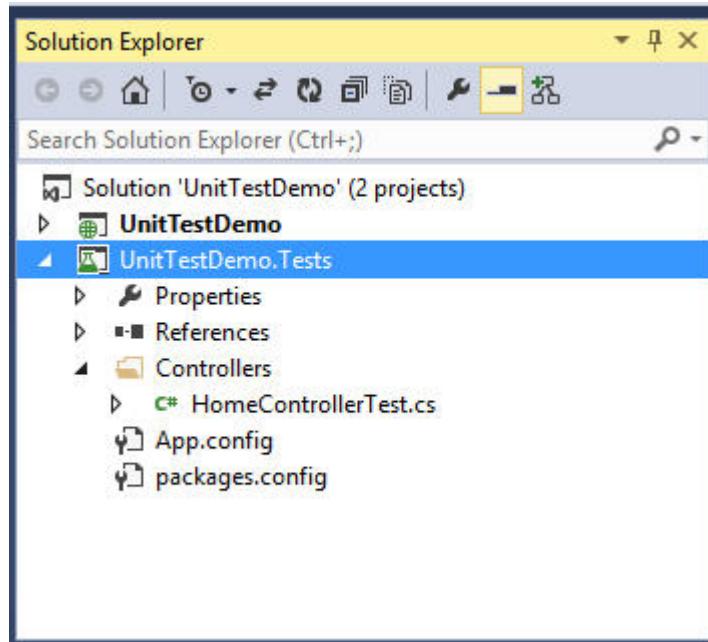


Figure 15.3: Displaying UnitTestDemo.Tests Node

5. Click the `HomeControllerTest.cs` under the **Controllers** folder to open it.

Note - The `HomeControllerTest.cs` file under the Controllers folder is created to test the Home controller of the application.

15.1.2 Unit Test Classes and Methods

A unit test class uses the `[TestClass]` attribute in the class declaration to indicate that it is a unit test class. Next, for each action method present in the target controller that needs to be tested, the unit test class has a corresponding method with the `[TestMethod]` attribute applied to it.

For example, to test an action method of the `Home` controller, the `HomeControllerTest.cs` file uses the `[TestClass]` attribute in the class declaration. Then, for each action methods of the `Home` controller, the `HomeControllerTest.cs` file uses the `[TestMethod]` attribute in the method declaration. A test method first creates an object of the `HomeController` controller class. Next, the test method retrieves the view of the action method as a `ViewResult` object. Finally, the `Assert.IsNotNull()` method is called passing the `ViewResult` object. The `Assert.IsNotNull()` method verifies that the `ViewResult` object passed as parameter is not null. If the call to the `Assert.IsNotNull()` method confirms that the passed `ViewResult` object is not null, the action method passes the unit test.

This is repeated for all the action methods of the controllers that returns `ViewResult` objects.

To test an action method that passes data to the view using a `ViewBag` object, the test method uses the `Assert.AreEqual()` method. This method accepts the message being passed to the view as the first parameter and `ViewResult.ViewBag.Message` property as the second parameter.

Code Snippet 1 shows the unit test class, named HomeControllerTest.

Code Snippet 1:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UnitTestDemo;
using UnitTestDemo.Controllers;

namespaceUnitTestDemo.Tests.Controllers
{
    [TestClass]
    public class HomeControllerTest
    {
        [TestMethod]
        public void Index()
        {
            // Arrange
            HomeController controller = new HomeController();
            // Act
            ViewResult result = controller.Index() as ViewResult;

            // Assert
            Assert.IsNotNull(result);
        }
        [TestMethod]
        public void About()
        {
            // Arrange
            HomeController controller = new HomeController();
            // Act
```

```
ViewResult result = controller.About() as ViewResult;

    // Assert
Assert.AreEqual("Your application description page.",
result.ViewBag.Message);
}

[TestMethod]
public void Contact()
{
    // Arrange
HomeController controller = new HomeController();

    // Act
ViewResult result = controller.Contact() as ViewResult;

    // Assert
Assert.IsNotNull(result);
}
}
```

This code uses the `[TestClass]` attribute in the class declaration and the `[TestMethod]` attributes for each action method that needs to be tested.

15.1.3 Performing Unit Tests

To unit test the Home controller class using the HomeControllerTest unit test, you need to perform the following step:

Select **Test→Run→All Tests** in Visual Studio 2013. The **Test Explorer** window displays the test results. Figure 15.4 shows the **Test Explorer** window.

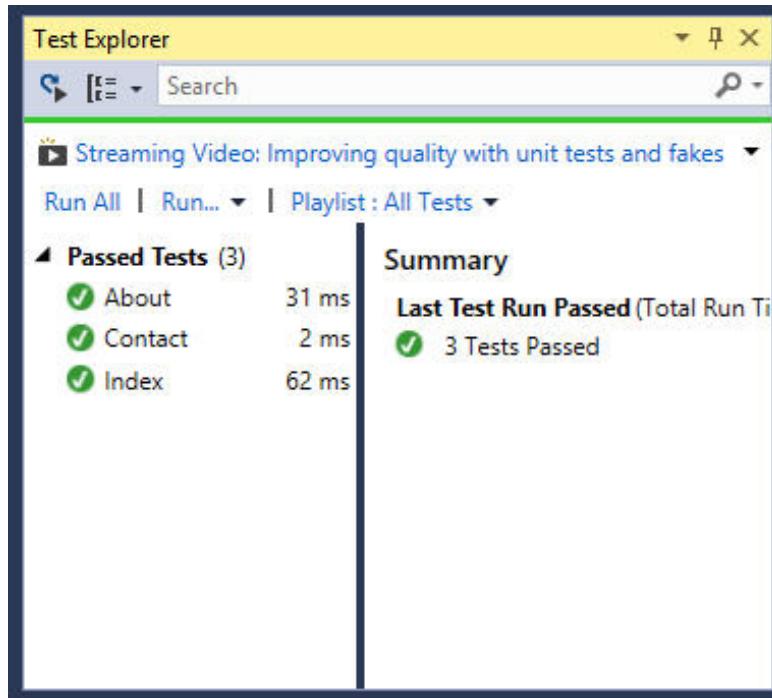


Figure 15.4: Test Explorer Window

15.2 Preparing the Application for Deployment

You have learned that after creating an ASP.NET MVC application, you should make it available to the users. The Internet is the best and the most used medium for getting information on any topic. To enable users to access the information contained in your application, you need to install the application on a Web server. The process of installing a Web application on a Web server is known as deployment.

While developing and executing an ASP.NET MVC application using Visual Studio 2013, the application automatically deployed on Internet Information Server (IIS) Express. IIS Express makes deployment simple, because it runs with your identity, and allows you to start and stop the Web server whenever required.

To make your application accessible over the Internet, you need to host it on an IIS or any other Web server.

Before deploying the application on a Web server, you first need to prepare the application for deployment. This process of preparing an application contains activities, such as identifying the files and folders to be copied on the Web server, configuring `Web.config` file, and precompiling the application.

15.2.1 Identifying the Files and Folders

When you create an application using Visual Studio 2013, the application is saved on the path of the local computer that you can explicitly specify. Once you have created the application then, you need to deploy it on a Web server, such as IIS. To deploy the application on IIS, first you need to identify the files and folders that are created in the specified path and need to be copied to the destination server.

While using Visual Studio 2013, by default, it deploys only those files and folders that are required to run the application. However, sometimes there might be a requirement where you need to copy several other files and folders to copy on the destination server. For example, consider a scenario where you only need some database files available in the `App_Data` folder while developing an application. In such scenario, you need to identify those files and exclude them while deploying the application. For this, you need to configure the deployment settings.

15.2.2 Configuring the `Web.config` File

Once you have deployed an ASP.NET MVC application on a Web server, some of the settings of the `Web.config` file vary in the deployed application. Consider a scenario, where you need to disable the debug options and change connection strings so that they point to different databases. You need to configure these types of settings in the `Web.config` transformation file, which is an XML file that allows you to specify how the `Web.config` file should be changed when it is deployed. You can specify such transformation actions in the `Web.config` file by using XML attributes.

A transform file is associated with a build configuration. When you compile and execute an application in Visual Studio 2013, by default, it creates the **Debug** and **Release** build configurations files named `Web.Debug.config` and `Web.Release.config` respectively. When you compile the application in Release mode, the `Web.release.config` file contains the changes that Visual Studio 2013 made in the `Web.config` file.

On the other hand, when you compile the application in the Debug mode, the `Web.debug.config` file contains the changes that Visual Studio 2013 made in the `Web.config` file.

To publish the application using release configuration, you need to remove the `debug` attribute from the `<compilation>` element in the `Web.config` file.

Code Snippet 2 shows how to remove the `debug` attribute.

Code Snippet 2:

```
<system.web>
<compilation xdt:Transform="RemoveAttributes(debug)" />
</system.web>
```

In this code, the `xdt:Transform` attribute is used to remove the `debug` attribute from the `Web.config` file.

15.2.3 Precompilation

To deploy an application, you need to copy the files and folders of the application to the hard drive of a Web server. In this process, most of the files are deployed to the Web server without compilation. Deploying an application in this way typically contains the following issues:

- ➔ The application might get deployed with compilation errors
- ➔ The source code of the application is exposed
- ➔ The application loads slowly because files are required to be compiled when it is accessed for the first time.

Precompiling a Web application is a process that involves compilation of the source code into DLL assemblies before deployment. The process of precompiling provides the following advantages:

- ➔ It provides faster response because the files of the application do not need to be compiled at the first time as it is accessed.
- ➔ It helps in identifying the errors that can occur when a page is requested, because errors are rectified at the time of compiling the application.
- ➔ It secures the source code of the application from malicious users.

15.3 Deploying on IIS

Consider the scenario where you have created an ASP.NET MVC application and it is ready to be made available to the user. To allow users to access the application, it has to be deployed on a Web server, such as IIS.

IIS is a Web server that allows you to develop, host, and manage your application. Before deploying an application on IIS, you first need to install it on your computer.

To deploy an ASP.NET MVC application, you need to perform the following tasks:

1. Install IIS
2. Create an application in IIS
3. Create a publish profile for the application
4. Publish the project

15.3.1 Installing IIS

The first step to deploy an application is to install IIS. To install IIS, you need to perform the following tasks:

1. Open **Control Panel**.
2. Click **Uninstall a program** under the **Programs** icon. The **Program and Features** window displays the **Uninstall or change a program** screen.
3. Click the **Turn Windows features on or off** link in the left pane. The **Windows Features** window is displayed.
4. Ensure that all the check boxes are selected under the **Internet Information Services** node. Figure 15.5 shows the **Windows Features** window.

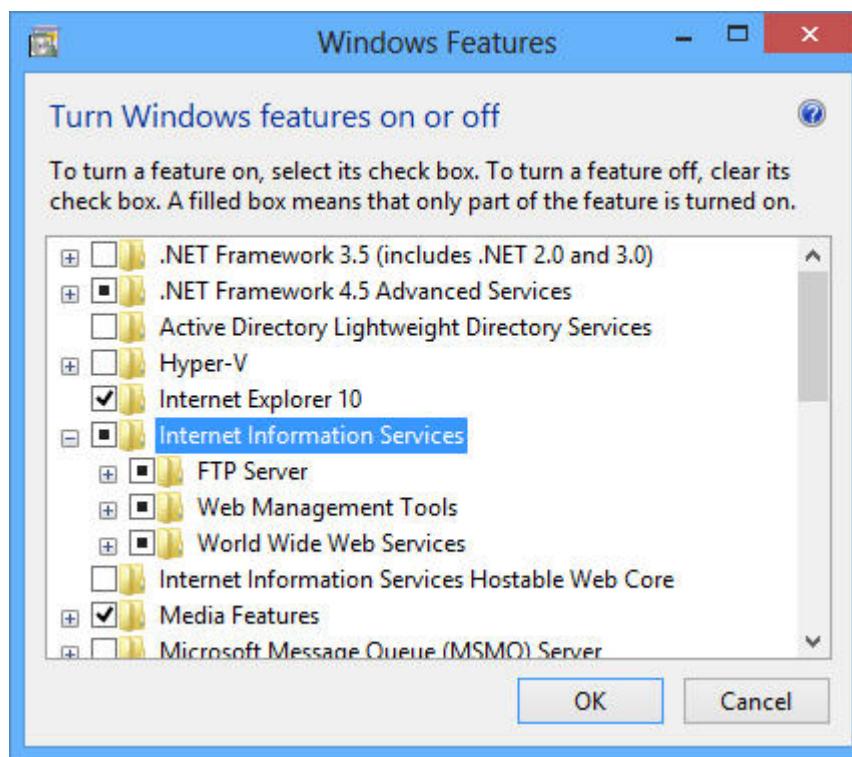


Figure 15.5: Windows Features

5. Click **OK**. The **Windows Features** message box is displayed.
6. Wait until the changes are applied to the system.
7. Click the **Close** button.
8. Close the **Programs and Features** window.

15.3.2 Creating an Application on IIS

Once you have installed IIS, the next step that you need to perform is creating a publish profile. To create an application on IIS, you need to perform the following steps:

1. Create a folder with the name of your project, for example, **MVCDemo** in the **C:\inetpub\wwwroot** folder.
2. Press the **Windows+R** keys. The **Run** dialog box is displayed.
3. Type **inetmgr** in the **Run** dialog box.
4. Click **OK**. The **Internet Information Services (IIS) Manager** window is displayed. Figure 15.6 shows the **Internet Information Services (IIS) Manager** window.

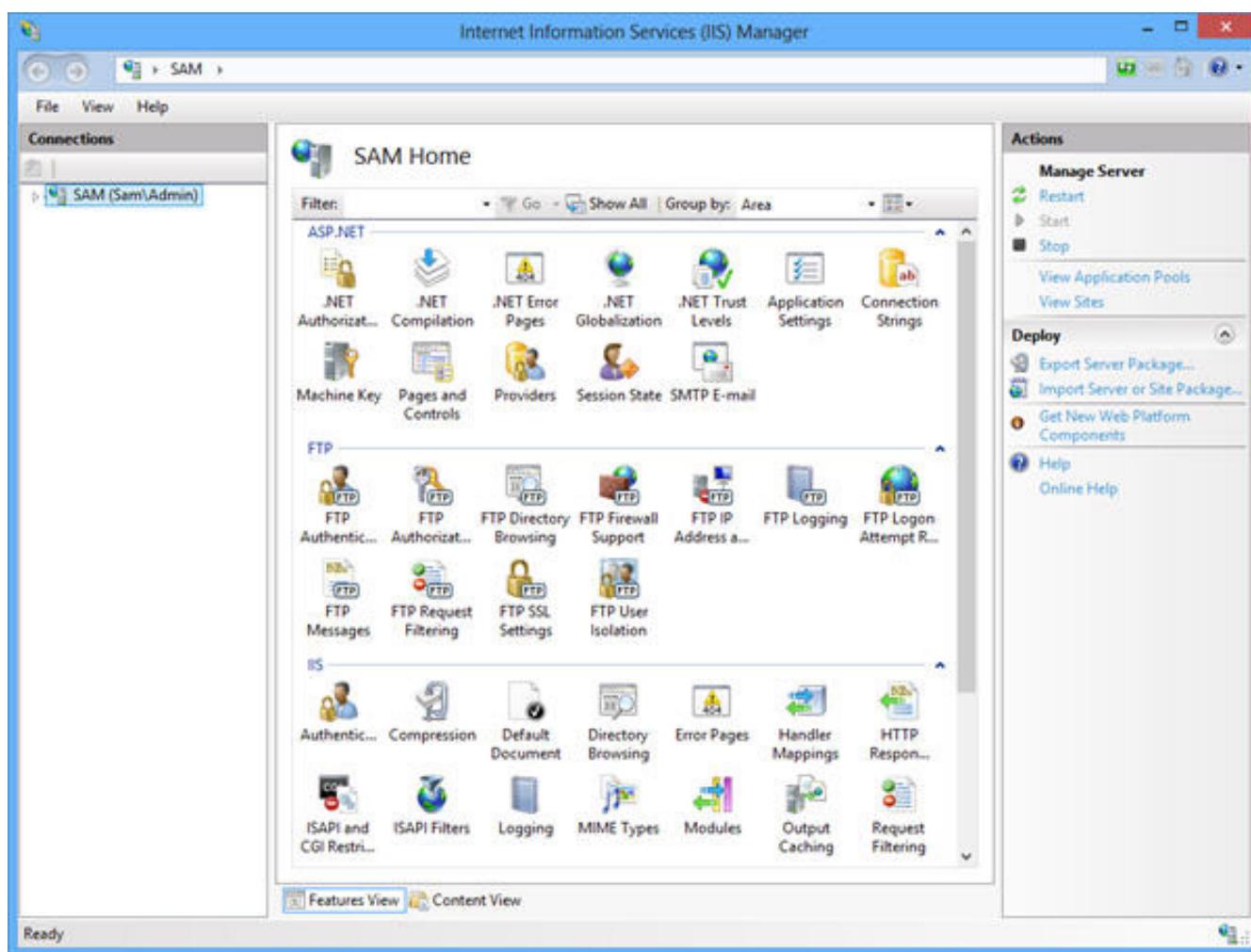


Figure 15.6: Internet Information Services (IIS) Manager Window

5. Expand the node under the **Connections** pane.
6. Expand the **Sites** node.
7. Right-click the **Default Web Site** node under the **Sites** node, and then, select **Add Application**. The **Add Application** dialog box is displayed.

8. In the **Add Application** dialog box, type **MVCDemo** in the **Alias** text field and type **C:\inetpub\wwwroot\MVCDemo** in the **Physical path** text field. Figure 15.7 shows the **Add Application** dialog box.

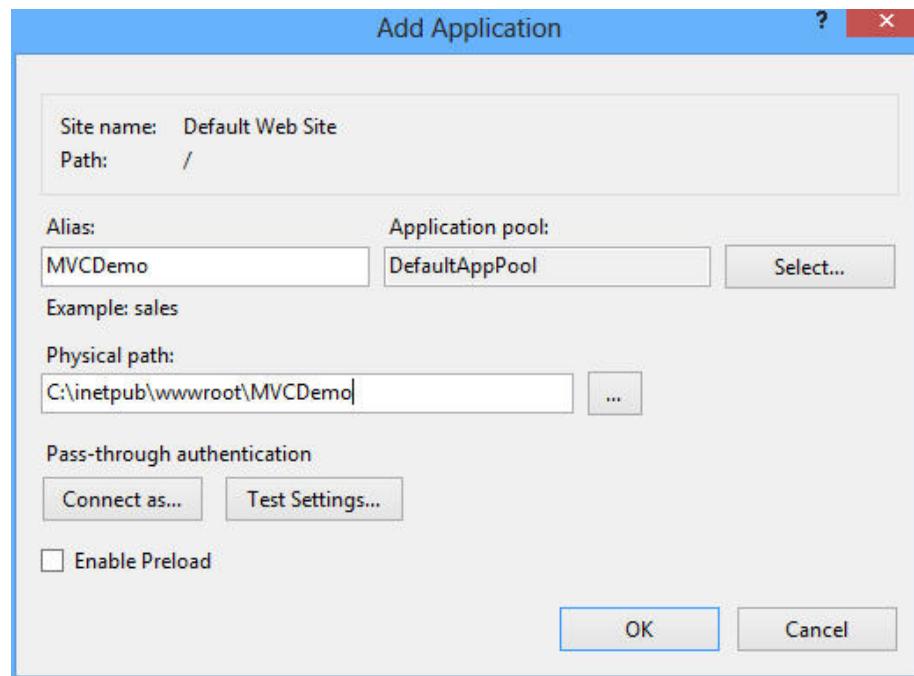


Figure 15.7: Add Application Dialog Box

9. Click the **OK** button in the **Add Application** dialog box. The **MVCDemo** node is displayed under the **Connections** node of the **Internet Information Services (IIS) Manager** window. Figure 15.8 shows the **MVCDemo** node under the **Connections** node.

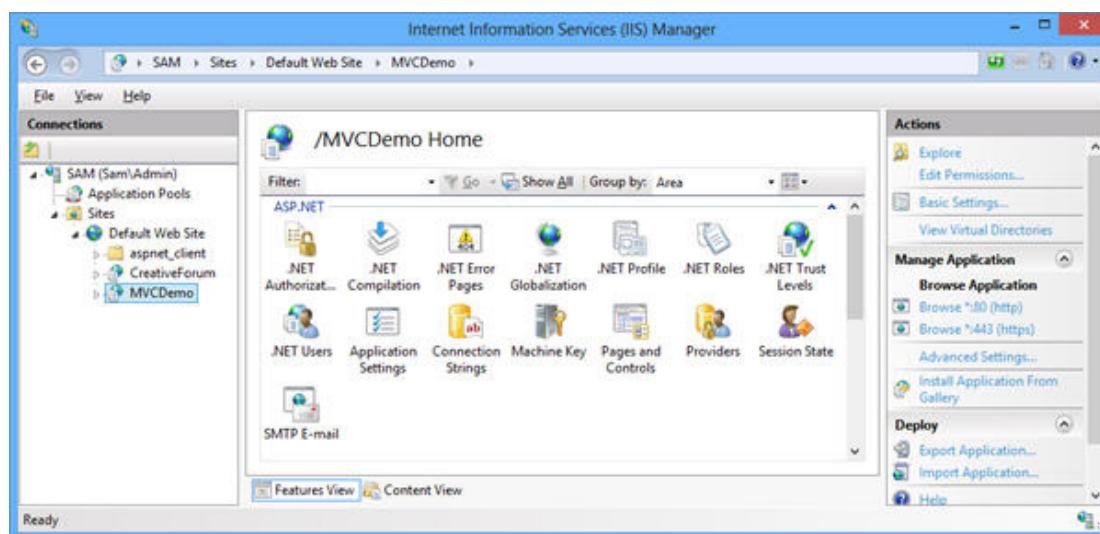


Figure 15.8: MVCDemo Node under the Connections

10. Close the **Internet Information Services (IIS) Manager** window.

15.3.3 Creating a Publish Profile

After creating the application on IIS, you need to create a publish profile. A publish profile represents various deployment options, such as the target server to be used for deployment, the credentials needed to log on to the server to deploy.

To create a publish profile in Visual Studio 2013, you need to perform the following tasks:

1. Start Visual Studio 2013 with Administrator privilege.
2. Open the `MVCDemo` project to publish.
3. Right-click the project in the **Solution Explorer** window and select **Publish**. The **Publish Web** dialog box is displayed.
4. From the **Select or import a publish** profile drop-down list, select the `<New...>` option.
5. The **New Profile** dialog box is displayed.
6. Type `MVCDemoPublishProfile` in the **Profile Name** text field. Figure 15.9 shows the **New Profile** dialog box.

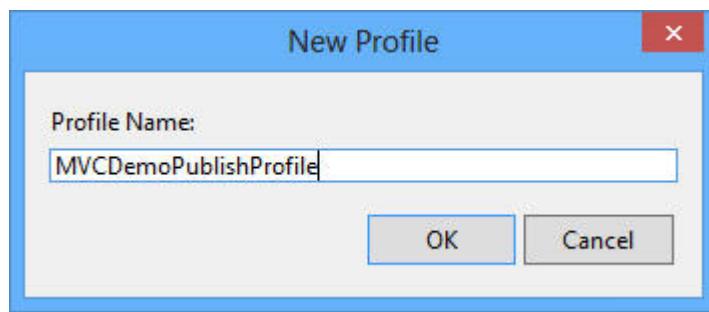


Figure 15.9: New Profile Dialog Box

7. Click **OK**. The **Publish Web** dialog box is displayed with the specified profile name.
8. Ensure that **Web Deploy** is selected in the **Publish method** drop-down list.
9. In the **Service URL** text field, type `localhost` and in the **Site/application** text field, type `Default Web Site/MVCDemo`.

Figure 15.10 shows the specifying the **Service URL** and **Site/application** text fields.

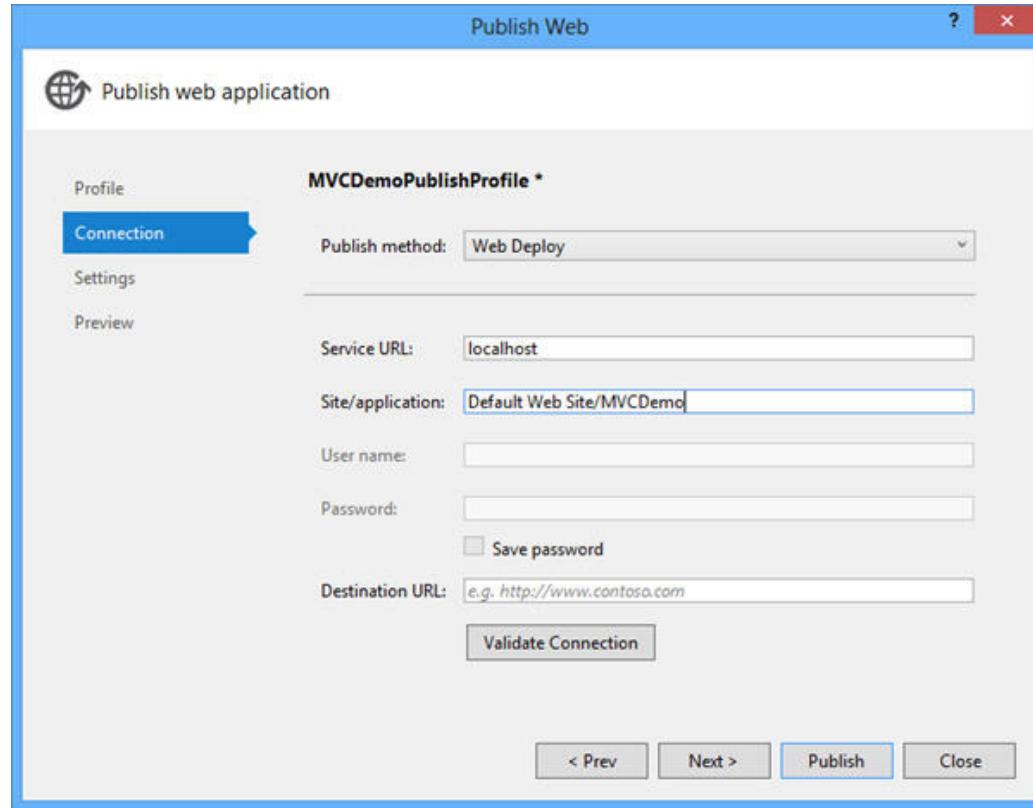


Figure 15.10: Publish Web Dialog Box

10. Click **Validate Connection**. When the connection is valid that is correct mark is displayed by the side of the **Validate Connection** button.

Figure 15.11 shows that the specified connection is valid.

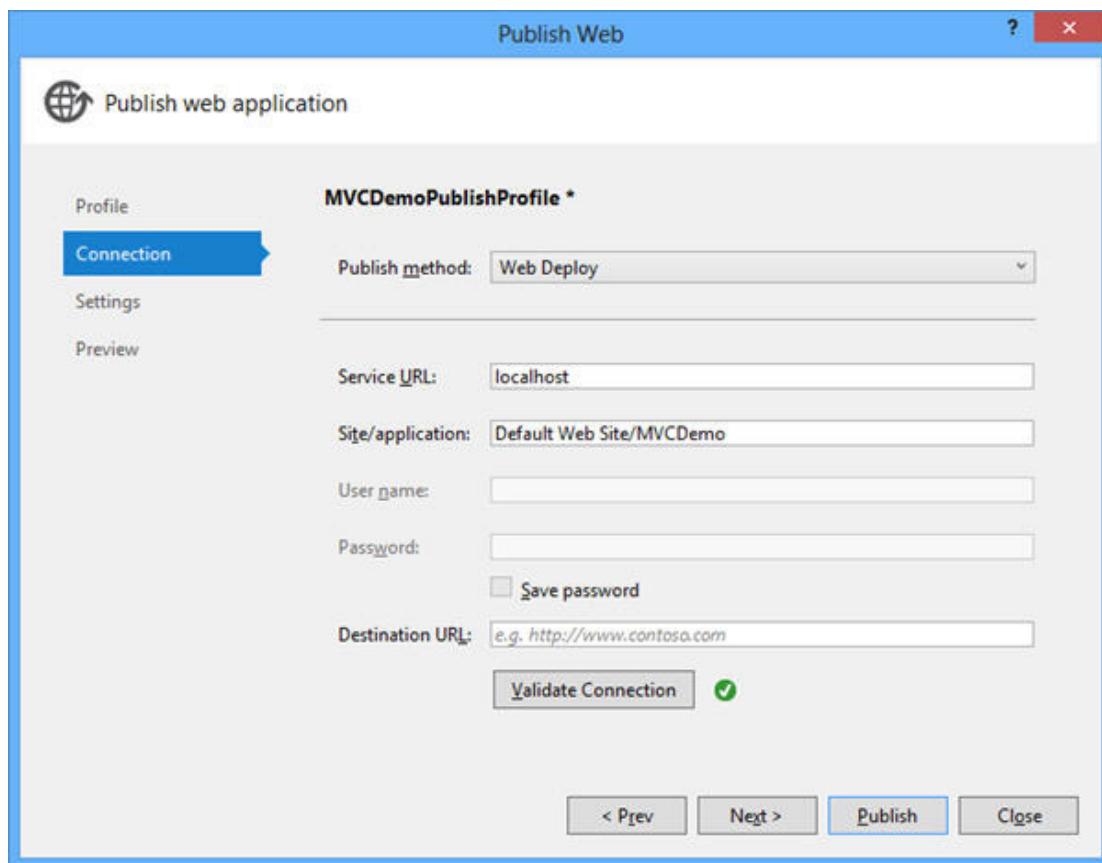


Figure 15.11: Displaying a Valid Connection

11. Click **Next**. The **Publish Web** dialog box is displayed.
12. Click **Publish**. The **Output** window of Visual Studio 2013 displays a message to indicate that the project has been successfully published. Figure 15.12 shows the **Output** window.

The screenshot shows the 'Output' window in Visual Studio. The title bar says 'Output'. The status bar at the bottom shows 'Show output from: Build'. The main area displays the following text:

```

1>COMMAND LINE: C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe C:\Users\...\\MVC Demo\...\MVC Demo.csproj /t:Publish
2>Adding file (Default Web Site/MVC Demo\Views\Shared\_LoginPartial.cshtml).
2>Adding file (Default Web Site/MVC Demo\Views\Web.config).
2>Adding file (Default Web Site/MVC Demo\Views\_ViewStart.cshtml).
2>Adding file (Default Web Site/MVC Demo\Web.config).
2>Adding ACL's for path (Default Web Site/MVC Demo)
2>Adding ACL's for path (Default Web Site/MVC Demo)
2>Publish is successfully deployed.
2>
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
===== Publish: 1 succeeded, 0 failed, 0 skipped =====
|
```

A red rectangular box highlights the line '2>Publish is successfully deployed.' and the summary lines at the bottom.

Figure 15.12: Output Window

To test the published application, type the following URL in the address bar of the browser:

`http://localhost/MVCDemo`

The Home page of the application published on IIS is displayed on the browser. Figure 15.13 shows the Home page of the published application on the browser.

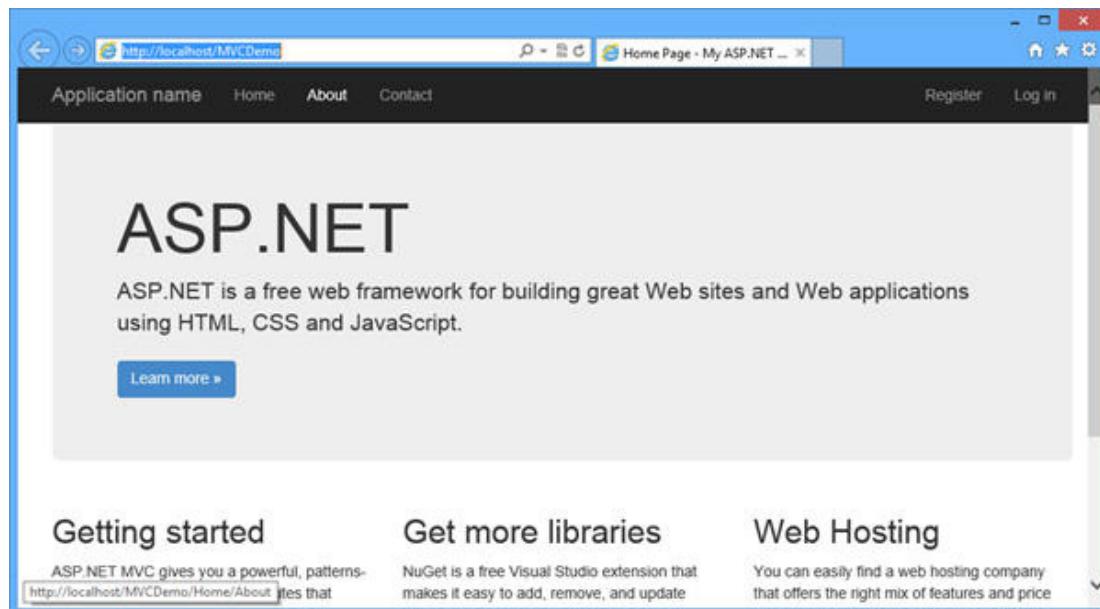


Figure 15.13: Home Page of the Deployed Application

15.4 Check Your Progress

1. Which of the following files you need to configure before deploying an application on a Web server?

(A)	Web.Debug.config
(B)	Global.asax
(C)	Web.config
(D)	Web.Release.config
(E)	Startup.cs

(A)	A	(C)	D
(B)	C	(D)	B

2. Which of the following files contain the changes that Visual Studio 2013 made when you compile an application in the Debug mode?

(A)	Web.Release.config
(B)	Web.config
(C)	Startup.cs
(D)	Global.asax
(E)	Web.debug.config

(A)	A	(C)	D
(B)	C	(D)	E

3. Which of the following directory structure on IIS you need to create for an application to deploy?

(A)	C:\inetpub\wwwroot
(B)	C:\inetpub\applicationname
(C)	C:\wwwroot\inetpub
(D)	C:\inetpub\wwwroot\deploy
(E)	C:\inetpub\deploy\wwwroot

(A)	B	(C)	A
(B)	C	(D)	E

4. Which of the following URL will you use to access an application that has been deployed on IIS server? Assuming that the name of your application is TestMVCApp.

(A)	http://inetpub/TestMVCApp
(B)	www.localhost/TestMVCApp
(C)	www.localhost/TestMVCApp/Home
(D)	https://localhost/TestMVCApp
(E)	http://localhost/TestMVCApp

(A)	B	(C)	D
(B)	E	(D)	A

5. Which of the following options represents various deployment options, such as the target server to be used and the credentials needed to log on to the server to deploy?

(A)	Publish profile
(B)	IIS profile
(C)	Unit testing
(D)	IIS manager
(E)	Web profile

(A)	B	(C)	A
(B)	C	(D)	E

15.4.1 Answers

(1)	B
(2)	D
(3)	C
(4)	B
(5)	C



Summary

- Unit testing is a technique that allows you to create classes and methods in your application and test their intended functionality.
- While developing and executing an ASP.NET MVC application using Visual Studio 2013, the application automatically deployed on Internet Information Server (IIS) Express.
- To deploy the application on IIS, first you need to identify the files and folders that are required to be copied on the destination server.
- While using Visual Studio 2013, by default, it deploys only those files and folders that are required to run the application.
- Precompiling a Web application is a process that involves compilation of the source code into DLL assemblies before deployment.
- To deploy an ASP.NET MVC application, you need to install IIS, create an application in IIS, create a publish profile for the application, and finally, publish the project.
- A publish profile represents various deployment options, such as the target server to be used for deployment, the credentials needed to log on to the server to deploy.