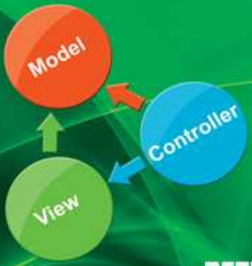
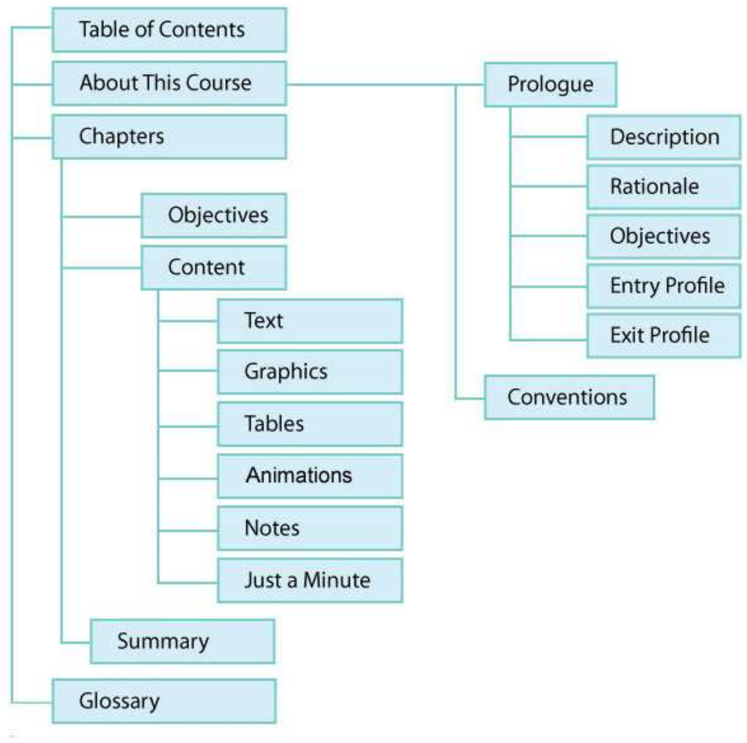
Student Guide

Developing Web Application Using ASP.NET MVC - Part I



NIIT

COURSE DESIGN - STUDENT GUIDE



Developing Web Application Using ASP.Net MVC SG-I /13-M08-V01

About This Course

Prologue

Description

ASP.NET provides a unified Web development model, which includes the services for creating websites easily and quickly. This course discusses various features, such as master pages and themes, which enable the developers to create pages that are more consistent and offer a richer experience to the users. It also discusses how to establish communication between a Web application and a database server. In addition, it discusses how to deploy Web applications on a server.

Rationale

With the increased use of the Internet and development in the field of Information Technology (IT), application developers need to quickly create applications that are accessible over the Web or a corporate intranet. These applications should also be efficient and effective.

ASP.NET provides developers with various time-saving and code-saving features. One of its key design goals is to make programming easier and quicker by reducing the amount of code. In addition, it contains several new server controls, which eliminate the need for writing voluminous code.

Objectives

After completing this module, the student will be able to:

- ❑ Identify the fundamentals of application development
- ❑ Work with Controllers, Views, Models, and Helper Methods
- ❑ Identify data annotations and implement validation
- ❑ Identify the fundamentals of Entity Framework
- ❑ Identify the fundamentals of LINQ
- ❑ Implement a consistent look and feel using layouts
- ❑ Make a Web application responsive by using JavaScript
- ❑ Implement the partial page updates using AJAX
- ❑ Implement state management and optimize the performance of a Web application
- ❑ Implement authentication and authorization
- ❑ Deploying a Web application

Entry Profile

The students who want to take this course should be:

- ❑ Familiar with fundamentals of computers.
- ❑ Familiar with programming logic and techniques.
- ❑ Familiar with HTML 5, CSS, and JavaScript.
- ❑ Able to interact in English in a classroom environment because the classes will be conducted in English.

Exit Profile

After completing this course, the student should be able to:

- ❑ Identify the various phases and development models of application development life cycle.
- ❑ Design interactive Web applications.

Conventions

Convention	*Indicates...*
📝	*Note*
🎞	*Animation*
⏰	*Just a Minute*

Chapter 1

Introduction to Web Application Development using ASP.NET MVC

Web applications have revolutionized the way business is conducted. These applications enable organizations to share and access information from anywhere, anytime. This has majorly moved the focus of application development from desktop applications to Web applications. Today, one of the most popular server-side technologies used for developing Web applications is ASP.NET.

This chapter introduces the basics of Web application development. It also discusses the layers, architecture, and different ways of scripting a Web application. In addition, it discusses the different types of Web development platforms and explains the ASP.NET MVC platform in detail.

Objectives

In this chapter, you will learn to:

- ❑ Identify the basics of Web application development
- ❑ Explore ASP.NET

Introduction to Web Application Development

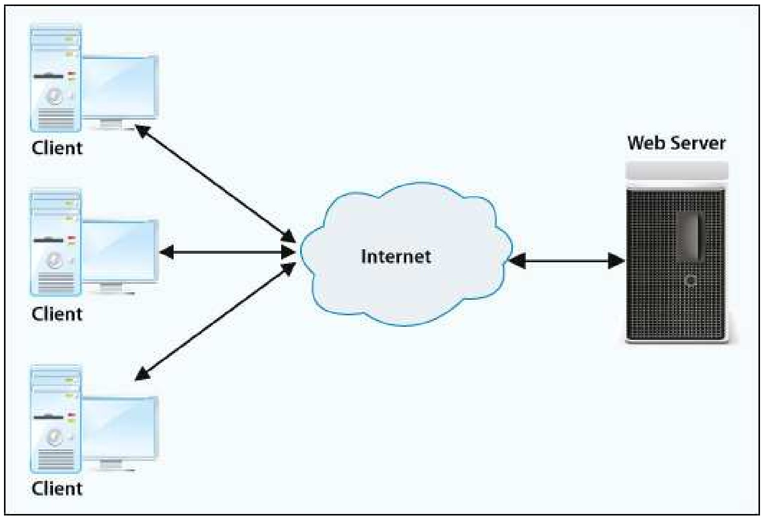
Among all technologies, the Internet has been the fastest growing technology. Ever since its inception, the Internet has evolved exponentially. In the recent years, it has changed the way business is conducted.

Prior to the evolution of the Internet, organizations could deliver only limited information to their prospective clients by using the existing communication media. However, with the inception of the Internet, organizations found a new medium to reach a larger range of people, irrespective of their geographical locations. Therefore, organizations increasingly became dependent on the Internet for sharing and accessing information. This resulted in changing the focus of application development from desktop applications to Web applications.

Defining Web Applications

Web applications are programs that are executed on a Web server and accessed from a Web browser. These applications enable organizations to share and access information on the Internet and corporate intranets. This information can be accessed from anywhere, anytime. In addition, Web applications can support online commercial transactions, popularly known as e-commerce. An online store accessed through a Web browser is an example of a Web application.

The following figure shows the working of a Web application.

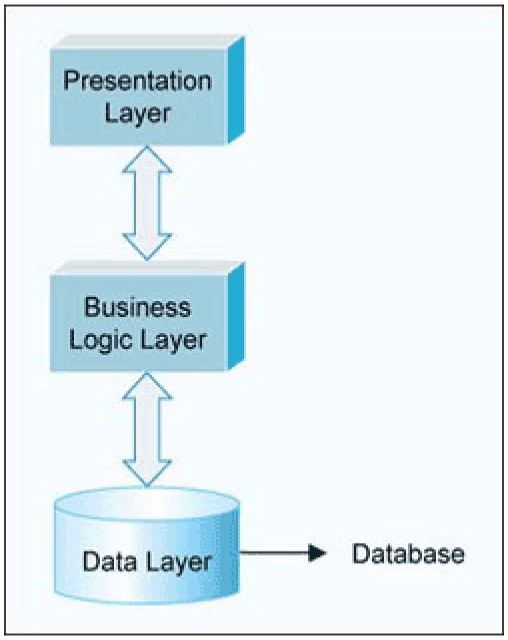


The Working of a Web Application

In the preceding figure, a client sends a request for a resource, such as a Web page or a video, on the Internet. The Web server interprets the client request and determines the type of resource requested by the client. If the required resource is found, the Web server sends the resource to the client. Otherwise, an error message is sent to the client.

The Three Layers of a Web Application

All applications can be broken into three layers. Each layer has its own components and functionality. The following figure depicts the various layers that constitute a Web application.



The Layers of a Web Application

As displayed in the preceding figure, an application has the following three layers:

- **Presentation layer:** Consists of the interface through which the users interact with the application.
- **Business logic layer:** Consists of the components of the application that control the flow of execution and communication between the presentation layer and the data layer.
- **Data layer:** Consists of components that expose the application data stored in databases to the business logic layer.

Architecture of a Web Application

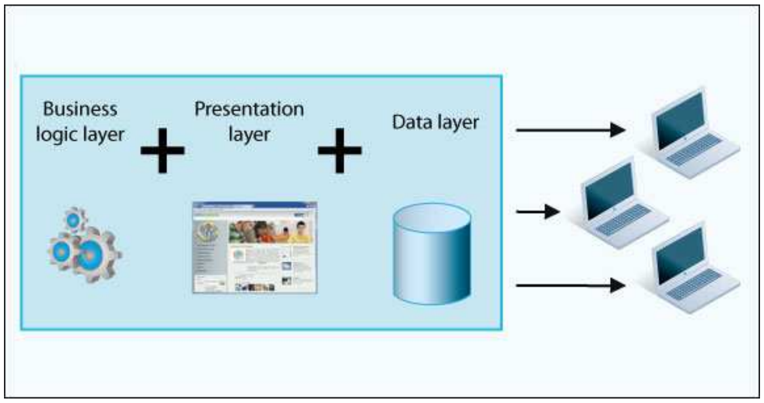
The architecture of a Web application is the manner in which layers are distributed and the way in which they communicate with each other. Most applications are built by using all the three layers.

An application can have one of the following types of architectures:

- Single-tier architecture
- Two-tier architecture
- Three-tier architecture
- N-tier architecture

Single-tier Architecture

In an application based on single-tier architecture, all the three layers are integrated together and can be installed on a single computer. If the application needs to be accessed on multiple computers, a separate installation is required on each computer. For example, Adobe Photoshop that is used to create and edit graphics is a standalone application based on the single-tier architecture. The following figure displays the single-tier architecture.



The Single-tier Architecture

Two-tier Architecture

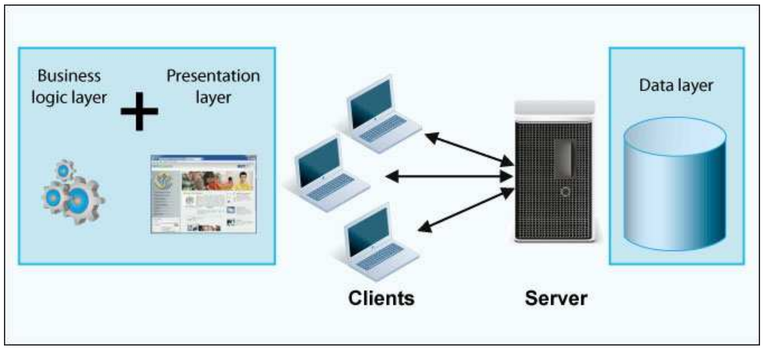
In an application based on two-tier architecture, the three layers are distributed over two tiers, a client and a server. The presentation layer resides on each client computer, the business logic layer resides either on the client or on the server, and the data access layer resides on the server.

Depending on the business requirements, an organization can have the following types of two-tier application architecture:

- **Fat client and thin server:** The architecture in which the business logic layer resides on the client is known as the *fat client and thin server architecture*. In this architecture, the client accepts user requests and processes these requests on its own. The client communicates with the server only when the data for communication or archival needs to be sent to the server.
- **Fat server and thin client:** The architecture in which the business logic layer resides on the server is known as the *fat server and thin client architecture*. In this architecture, the client accepts requests from the users and forwards the same to the server. Further, the server processes these requests and provides responses.

The two-tier architecture can be used when multiple users need to access and manipulate common data storage. For example, an application needs to be developed to store an organization's employee details in a database. In addition, the application should allow the retrieval of employee details from the database. In this case, the employee details need to be entered and retrieved by using multiple computers. Therefore, installing a database, along with the application on each computer, may lead to storage of duplicate records. In addition, a user cannot retrieve the employee details stored on some other computer. To avoid these problems, the application can be installed on each computer. However, the database is installed on a single computer, and the same can be accessed by the applications on multiple computers.

In the preceding example, the presentation and business logic layers are integrated and installed on each computer. The applications installed on the computers send requests to the server (database) for the storage and retrieval of data. On the other hand, the server accepts the requests and responds accordingly. In this setup, the fat client and thin server architecture is used. The following figure displays the fat client and thin server architecture.

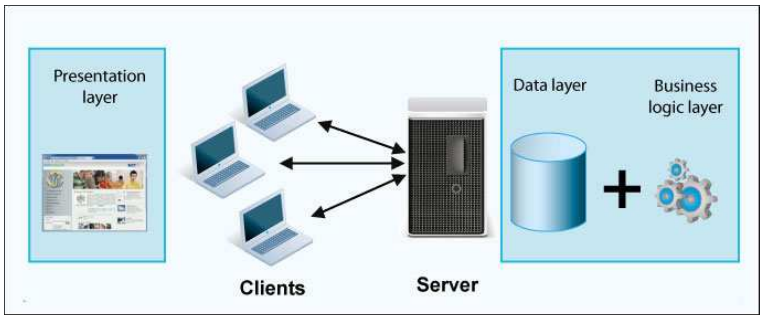


The Fat Client and Thin Server Architecture

In the preceding setup, the server only processes the data processing requests sent by the clients. Therefore, data retrieval is fast. As the business logic resides on the client, the other applications installed on the clients respond slowly. Further, if the business logic needs to be updated, the updates need to be installed on each client, which is a time-consuming process.

To overcome these limitations, the business logic layer can be placed on the server along with the data layer. In this way, only the presentation layer will be available on the clients. This, in turn, results in a faster response from the other applications installed on the clients. In addition, if the business logic needs to be updated, the updates need to be installed at only one location, which is on the server, rather than on each client. Such a setup is an example of the thin client and fat server architecture.

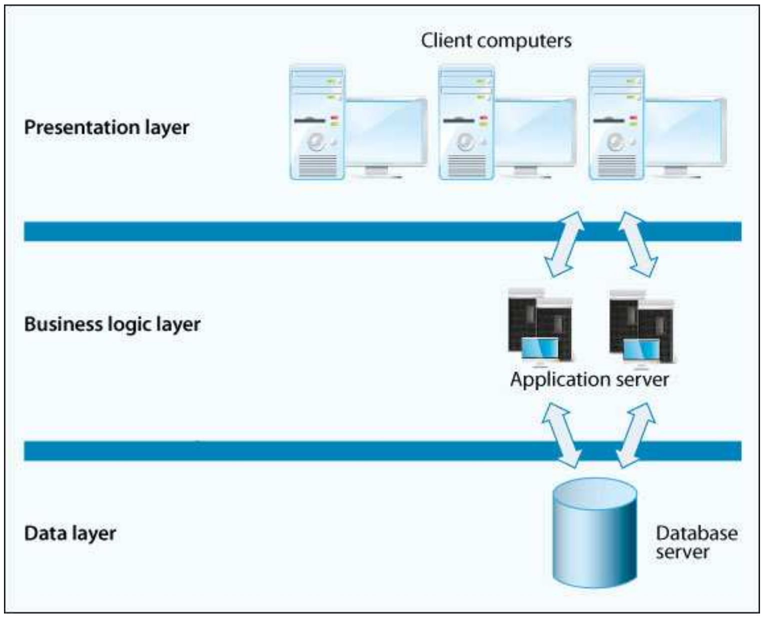
The following figure displays the thin client and fat server architecture.



The Thin Client and Fat Server Architecture

Three-tier Architecture

In an application based on three-tier architecture, the three layers of the application are placed separately as three different entities. This architecture is used for those applications in which merging the business logic layer with the presentation layer or the data layer may degrade the performance of the application. To improve the application performance, the three layers are kept separately and the communication among the layers occurs with the help of a request-response mechanism. The following figure displays the three-tier architecture.



The Three-tier Architecture

In the preceding figure, the business logic layer resides neither on a client nor on a database server. Instead, it resides on a separate server, which is known as an application server.

N-tier Architecture

The three-tier architecture is implemented when an effectively-distributed client/server architecture, which provides increased performance and scalability, is needed. However, it becomes a tedious task to design and deploy applications based on the three-tier architecture. The separation of presentation, business logic, and database access layers is not always obvious, because some business logic needs to appear on all the layers. Therefore, there is a need to further divide these layers. This requirement has led to the expansion of the three-tier application architecture to the N-tier application architecture.

A common approach to implement the N-tier architecture involves further separation of the presentation layer and the data layer. For example, the presentation layer may be divided into the GUI layer and the presentation logic layer. Similarly, the data layer may be divided into the data access layer and the data layer.

The N-tier architecture separates the various business processes into discrete units of functionality called services. Each service implements a set of related business rules that defines what needs to be done and how it needs to be done within an organization.

The N-tier application architecture provides improved flexibility and scalability of applications, as compared to the three-tier application architecture. This is because the functioning of each layer is completely hidden from other layers, which makes it possible to change or update one layer without recompiling or modifying the other layers.

Introduction to Web Pages

A Web application consists of certain Web pages. A Web page is a Web document that can be accessed through a Web browser. Web pages can be of the following types:

- **Static Web page:** A Web page that contains only static content and is delivered to the user as it is stored is called a *static Web page*. A static page does not offer any interactivity to the user.
- **Dynamic Web page:** A Web page whose content is generated dynamically by a Web application or that responds to user input and provides interactivity is called a *dynamic Web page*.

Creating a Dynamic Web Page

To create a dynamic and interactive Web page, you need to use scripting, which includes:

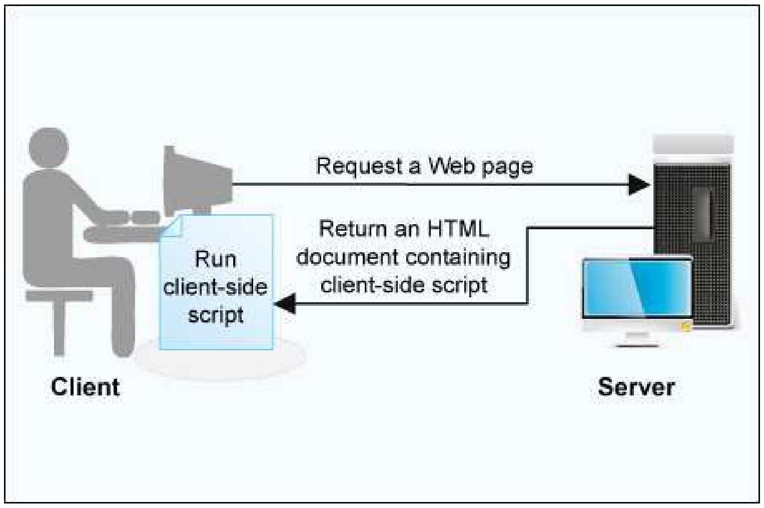
- Client-side scripting
- Server-side scripting

Client-side Scripting

Client-side scripting enables you to develop Web pages that can dynamically respond to user input without having to interact with a Web server. Suppose you have a Web application that requires users to enter the user name and password before displaying the home page. To check whether the user name and password provided by the user are correct, the details need to be submitted to the server. However, before submitting the details to the server, you need to ensure that the user does not leave the user name and password fields blank. To check whether the user has left the user name and password fields blank, you can write a client-side script.

In addition to providing dynamic content, a client-side script helps in reducing the network traffic because it does not need to interact with a Web server to provide a dynamic response to the user input. Client-side scripting also speeds up the response time of a Web application. This happens because the processing happens at the client side and round-trips to the server are not required. Scripting languages, such as VBScript and JavaScript, are used to write client-side scripts.

The following figure displays the working of client-side scripts.



The Working of Client-side Scripts

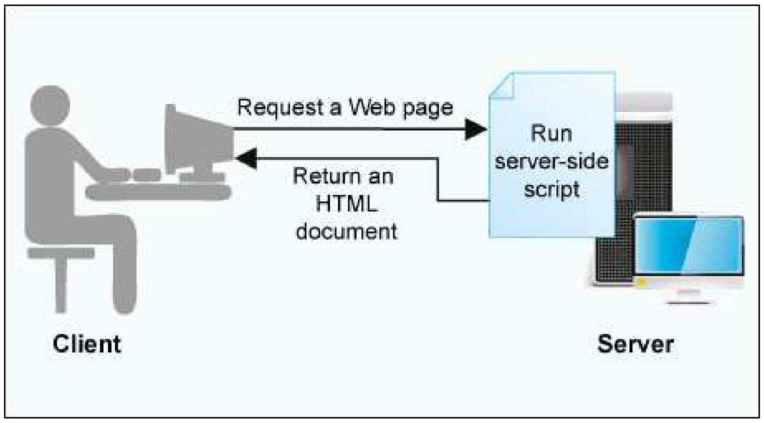
Server-side Scripting

With the growing usage of the Internet as a medium of information, organizations are increasingly creating dynamic Web applications. Server-side scripting helps in making a Web application dynamic by enabling the application to access server-side resources, such as files and databases.

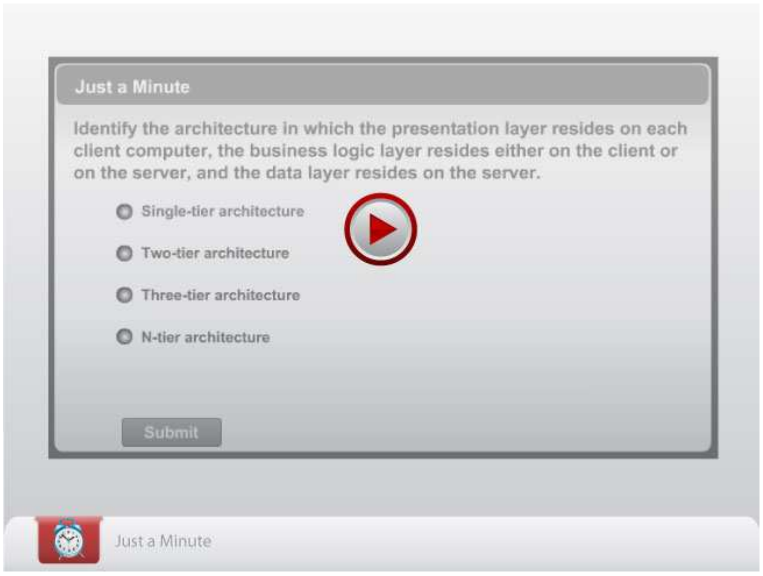
Server-side scripting provides users dynamic content that is based on the information stored at a remote location, such as a back-end database. Server-side scripting includes code written in server-side scripting technologies, such as Active Server Pages (ASP) and Java Server Pages (JSP).

A server-side script is executed on the Web server. When a browser requests for information on a website that is created by using the server-side technology, the Web server, to which the request is sent, processes the script, and then sends the results back to the browser. For example, if a Web page includes a server-side script to display the current time of the system on which the website is hosted, the script will be processed at the server and the current system time will be sent back to the browser.

The following figure displays the working of server-side scripts.



The Working of Server-side Scripts



Just a Minute

Identify the architecture in which the presentation layer resides on each client computer, the business logic layer resides either on the client or on the server, and the data layer resides on the server.

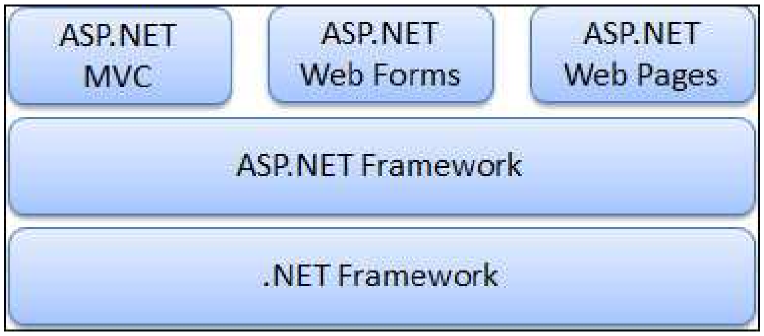
- Single-tier architecture
- Two-tier architecture
- Three-tier architecture
- N-tier architecture

Exploring ASP.NET

ASP.NET is a server-side technology that enables programmers to create dynamic Web applications. It has a number of advanced features, such as simplicity, security, and scalability, which help you in developing robust Web applications. These advanced features of ASP.NET are based on .NET Framework.

Introducing Microsoft's Web Development Platforms

ASP.NET is built on Microsoft .NET Framework. The following figure depicts ASP.NET in the .NET Framework.



ASP.NET in the .NET Framework

As displayed in the preceding figure, ASP.NET is a Web application development framework built on top of Microsoft's .NET Framework, which provides various components, such as CLR and the base class library, to develop robust applications. Common Language Runtime (CLR) is the runtime environment provided by .NET framework, which manages code at runtime and provides core services, such as multiple language support and strong type checking. The base class library is an object-oriented class library that provides a set of classes, which enables you to perform a wide range of common programming tasks. These tasks include database connectivity, file access, and string management. In addition to these common programming tasks, the base class includes the classes that support specialized functions, such as developing console, GUI, and Web applications.

ASP.NET works on top of the .NET Framework and allows you to access the classes in .NET Framework. ASP.NET interacts with the .NET Framework base classes, which, in turn, interact with CLR. CLR helps you to code your ASP.NET applications in any language compatible with it including Microsoft Visual Basic and C#. This provides a robust Web-based development environment.

ASP.NET offers various Web application development models. These are:

- ❑ Web forms
- ❑ Model-View-Controller (MVC)
- ❑ Web pages

Each model creates and structures a Web application by using different approaches. You can choose any of these models depending on the requirement of the Web application to be developed.

Web Forms

The ASP.NET Web forms model enables you to develop dynamic and powerful websites. It provides many controls, such as text boxes and check boxes, which help you to build an attractive user interface with data access. It enables you to drag and drop server controls to design your Web forms pages. You can easily set the properties, methods, and events for a control or a page to specify the page's behavior. Web forms are pages that are written by using a combination of HTML, server controls, and 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.

Server controls are those which are processed at the server.

Model-View-Controller (MVC)

The MVC model is an alternative to the ASP.NET Web forms model for developing Web applications. It is a light-weight model and supports existing ASP.NET features. It separates the following three main components from an application:

- **Model:** Refers to a set of classes that describes the data that the application works with. In addition, these classes define the business logic that governs how the data can be manipulated.
- **View:** Refers to the components that define an application's user interface. For example, the login form that contains text boxes and buttons.
- **Controller:** Refers to a set of classes that handle communication from the user and the overall application flow. A controller responds to user input, communicates with the model, and decides the view to render.

By separating the model, view, and controller within an application, the MVC model ensures loose coupling. Therefore, it enables you to focus on one aspect of the implementation at a time. For example, you can concentrate on designing a view without worrying about the business logic.

Loose coupling refers to an approach of interconnecting the components in such a way that components depend on each other to the least extent.

The MVC model makes it easier to manage complexity by dividing an application into the model, the view, and the controller. Therefore, it should be used for developing Web applications that are managed by large teams of developers and Web designers.

Web Pages

Web pages are the simplest way to develop dynamic ASP.NET Web applications. The Web pages combine HTML, CSS, JavaScript, and server code at one place to build dynamic websites. The Web pages framework uses inline scripts rather than separate controller classes.

Introduction to the MVC Development Platform

Web applications usually need to retrieve data from a database and display it to the user. In addition, Web applications may also need to update the data in the database if the user changes the data using the User Interface (UI). Since the key flow of information is between the database and the user interface, you might want to bind these two pieces together. However, the user interface tends to change much more frequently than the database.

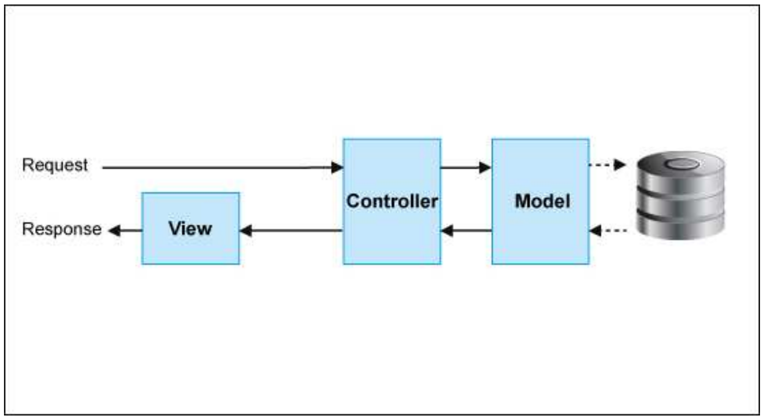
If the interface and the database logic are combined in a single object, then you need to modify the database logic whenever the user interface is changed. This can result in errors, and therefore, requires the retesting of the entire application.

The MVC development platform overcomes this limitation. It separates the application data, the presentation, and the flow of application, into three separate components: the model, the view, and the controller. The model manages the application data. It encapsulates data stored in a database as well as code used to manipulate the data and enforces domain-specific business logic. The view manages the display of information. The controller manages the flow of application, handles user input, and provides data to the relevant view. This separation of view, model, and controller within an application allows you to change one component without affecting the other component.

The ASP.NET Implementation of MVC

In MVC, incoming requests are handled by controllers. Controllers are C# classes that contain methods. Each public method in a controller is known as an action method or a controller action. These action methods can be invoked by using a URL.

When an HTTP request is sent to the server by using a URL, the controller action associated with the URL is executed in order to perform the required operation on a model. Then, a view is rendered and displayed to the client. The following figure displays the interactions between the controller, the model, and the view.



The Interactons between the Controller, the Model, and the View

In the preceding figure, the HTTP request is sent to the controller. The controller works with the model, selects the view, and finally, prepares the response. This response is rendered by the selected view and is sent to the client.

Features of ASP.NET MVC

Some of the features of ASP.NET MVC are:

- **Routing:** MVC has a powerful routing system. *Routing* is a feature that enables you to develop applications with comprehensible and searchable URLs. When you create the ASP.NET MVC application, the application is already configured to use ASP.NET routing. Routing enables you to use URLs that do not have to map to specific files in a website. Therefore, you don't need to include file names in URLs. You need to write URLs that provide the description of the user's action. Therefore, they can be easily understood by the users.
- **Scaffolding:** MVC provides a scaffolding feature that provides a quick way to generate the code for commonly used operations in a standardized way. It creates the essential working code automatically that you can edit and adapt according to your needs.
- **Convention-over-Configuration:** ASP.NET MVC supports Convention-over-Configuration, which is a software design paradigm that aims at reducing the number of decisions taken by developers. Therefore, instead of requiring the developers to provide explicit configuration settings, ASP.NET MVC assumes that developers will follow certain conventions while building their applications. Some of the conventions are:
  - The name of each controller's class should end with `Controller`. For example, `CustomerController` and `AccountController`.
  - All the views of an MVC application should be placed in the Views folder.
  - Views are stored in a subfolder of the Views folder, where the name of the subfolder is same as the name of the relevant controller. For example, all the views for the `AccountController` controller will be placed in the folder, /Views/Account.
- **Bundling and Minification:** *Bundling* is a technique provided by ASP.NET MVC that allows you to combine multiple files, such as CSS and JavaScript, into a single file. Bundling improves the request load time by reducing the number of requests to the server. Another technique, minification, removes unnecessary white space and comments from the JavaScript or CSS code, and shortens variable names. This helps in minimizing the size of the requested resources, such as CSS and JavaScript files.

Key Benefits Provided by ASP.NET MVC

ASP.NET MVC helps in developing applications in a loosely coupled manner, which means that different components of an MVC application depend on each other to the least extent practicable.

Limiting interconnections between these components helps in isolating problems, which provides the following benefits:

- **Separation of concerns**: Loose coupling ensures dividing various application concerns into different and independent software components. This allows developers to work on one component independently at a time.
- **Simplified testing and maintenance**: You can test each component separately to ensure that it is working according to the requirements. This simplifies testing, maintenance, and troubleshooting procedures.
- **Extensibility**: The MVC model consists of a series of independent components. These components can be easily replaced or customized depending upon the requirements, without hampering the functionality of the application.

Structure of an ASP.NET MVC Project

When you create an ASP.NET MVC Web application project, separate folders are created for the MVC components, such as model, view, and controller. Some of the project folders that are created by default are listed in the following table.

Project Folders	***Description***
App_Data	*It is the location for storing data files.*
Content	*It is the recommended location for adding content files, such as images and cascading style sheets.*
Controllers	*It is the recommended location to add controllers.*
Models	*It is the recommended location for adding classes that represent the application model.*
Scripts	*It is the recommended location for adding script files, such as jQuery and JavaScript files. By default, this folder contains ASP.NET AJAX foundation files and the jQuery library.*
Views	*It is the recommended location for adding views. The Views folder contains a folder for each controller, which is named with the controller-name prefix. For example, if you have a controller named AccountController, then the Views folder contains a folder named Account, which contains the views associated with the AccountController controller.*

The Project Folders

In addition to the folders listed in the preceding table, an MVC application can include different types of files. The following table lists some of the important file types with their descriptions.

File Name	***Description***
Ends with .cshtml	*These files contain the user interface code.*
Web.config	*This is an XML-based configuration file for ASP.NET MVC applications. It includes settings for customizing features, such as security, state management, and memory management.*
Global.asax	*This is the global application file. You can use this file to define global variables (variables that can be accessed from any Web page in the Web application). It is usually used for defining the overall application events.*
Ends with .cs	*These files contain C# code.*

The Files Used in the MVC Application

ASP.NET uses a hierarchy of configuration files to keep application configuration settings separate from the application code. A configuration file is an XML file that contains configuration settings for an application and has a .config extension.

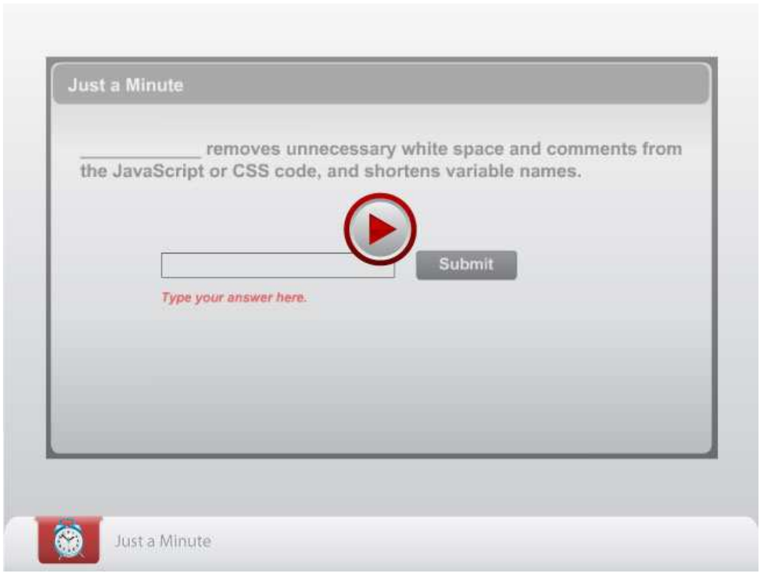
A configuration file provides the following benefits:

- ❑ Provides control and flexibility over the way you run applications.
- ❑ Eliminates the need to recompile the application every time a setting changes.
- ❑ Controls access to the protected resources and the location of remote applications and objects by defining configuration settings.

The two files included in the hierarchy of configuration files are Machine.config and Web.config. The Machine.config file always resides at the root of the configuration hierarchy and contains the global and default settings for all the .NET Framework applications on the server. The Web.config file can reside at multiple levels in the hierarchy. In case two versions of the same configuration setting exist at different hierarchal levels, the local settings will take precedence over global settings. The following table lists the configuration files, their hierarchy levels, and description.

Level	***File Name***	***Description***
Machine config	*Machine.config*	*Controls the basic settings of all .NET applications that are on the same computer.*
Machine Web.config	*Web.config*	*Controls the settings of all ASP.NET applications on the same machine.*
Root Web.Config	*Web.config*	*Controls the settings of all applications under a parent application. This level would exist only if your application is deployed under another ASP.NET application.*
Application Web.config	*Web.config*	*Contains the settings that are specific to an application and is present in the root directory of each application.*

The Configuration Files with their Levels and Descriptions


Just a Minute

__________ removes unnecessary white space and comments from the JavaScript or CSS code, and shortens variable names.

Task 1.1: Exploring an ASP.NET MVC Project

Summary

In this chapter, you learned that:

- Web applications are programs that are executed on a Web server and accessed from a Web browser.
- An application can have one of the following types of architectures:
  - Single-tier architecture
  - Two-tier architecture
  - Three-tier architecture
  - N-tier architecture
- A Web page is a Web document that can be accessed through a Web browser.
- To create a dynamic and interactive Web page, you need to use scripting, which includes:
  - Client-side scripting
  - Server-side scripting
- ASP.NET is a server-side technology that enables programmers to create dynamic Web applications.
- ASP.NET offers various Web application development models. These are:
  - Web forms
  - Model-View-Controller (MVC)
  - Web pages
- Controllers are C# classes that contain methods. Each public method in a controller is known as an action method or a controller action.
- Some of the features of ASP.NET MVC are:
  - Routing
  - Scaffolding
  - Convention-over-Configuration

    - Bundling and Minification

- ASP.NET uses a hierarchy of configuration files to keep application configuration settings separate from the application code.
- A configuration file is an XML file that contains configuration settings for an application and has a .config extension.

Chapter 2

Working with Controllers and Views

The purpose of many Web applications is to retrieve data from a database and display the same to the user. Web applications may also need to update the data in the database if the user makes any change in the data using the User Interface (UI). Since the flow of information takes place between the database and the user interface, you might want to bind these two pieces together. However, because the user interface tends to change much more frequently than the database, the approach of binding is not feasible.

The MVC pattern solves this problem by separating the application data, the presentation, and the flow of application, into three separate components: the model, the view, and the controller. This separation allows you to individually manage each component without affecting the functionality of other components.

This chapter discusses how to work with controllers and views. Further, it discusses how to route requests to controller actions.

Objectives

In this chapter, you will learn to:

- ❑ Work with controllers
- ❑ Work with views
- ❑ Route requests to controller actions

Working with Controllers

An MVC application consists of three parts: model, view, and controller. The development of a Web application usually requires the knowledge of all the three parts. Moreover, it is difficult to dive deep while exploring any of these parts without knowing about the others. Let us first learn about controllers at a high level, while ignoring views and models for the time being.

Controllers within an MVC application are responsible for processing incoming user requests, executing the appropriate application code, communicating with the model, and rendering the required view. In other words, controllers manage the flow of the application.

Controllers are implemented in an MVC application as C# classes inherited from .NET Framework's built-in `System.Web.Mvc.Controller` class. A `controller` class contains an application logic in the form of various public methods called action methods.

Creating a Controller

In ASP.NET MVC, controllers handle all the incoming requests. By convention, controllers are placed in a folder named Controllers under the Web application folder. Thus, for creating a controller, you need to create a **.cs** file containing the controller class in the Controllers folder. You create a controller class by inheriting the `Controller` class, as shown in the following code snippet:

```
public class HomeController :
Controller
{
 //Some code
}
```

In the preceding code snippet, a controller class named `HomeController` is created, which inherits the `Controller` class.

According to the ASP.NET MVC convention, the name of a controller is always suffixed by the word, Controller. For example, the name of the Home controller will be the HomeController.

Creating Action Methods

An application needs to handle user interactions, such as entering a URL in the Web browser, clicking a hyperlink, and submitting a Web form. Each of these interactions sends a request to the Web server. In ASP.NET MVC, such requests are handled by public methods in the controllers. These public methods are called action methods or controller actions. You can define these action methods in a controller class. For example, the following code snippet defines two action methods, `Index` and `About`, in the `HomeController` class:

```
public class HomeController :
Controller
{
 public ActionResult Index()
 {
   //Some code
 }
public ActionResult About()
 {
  //Some code
 }
}
```

In the preceding code snippet, two action methods, `Index` and `About`, are created in the `HomeController` controller class. Both these methods return an instance of the `System.Web.Mvc.ActionResult` class, which is the base class of all action results.

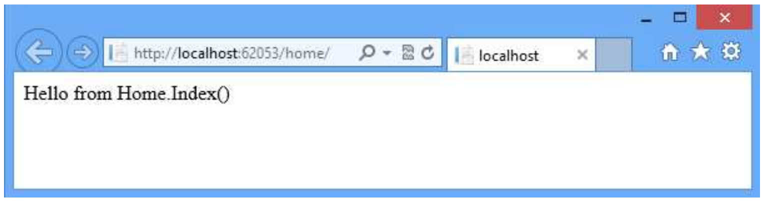
Most action methods return an instance of a class that is derived from the `ActionResult` class. However, you can also use specific return types, such as a `string`, `int`, or `bool`, as shown in the following code snippet:

```
public class HomeController :
Controller
 {
   public string Index()
   {
       return "Hello from Home.Index
 ()";
   }
 }
```

In the preceding code snippet, an action method, `Index()`, is created in the `HomeController` controller class. Here, the return type of the `Index()` action method is string, instead of the `ActionResult`. When a user invokes the preceding `Index()` method using the **URL, http://localhost:62053/home/index**, the `Index()` action method sends the string, **Hello from Home.Index()**, to the browser.

*In the preceding URL **http://localhost:62053/home/index**, 62503 is a port number. The port number is automatically generated and may vary from project to project.*

The following figure shows the response of the `Index()` action method in a browser.



Hello from Home.Index()

The Response of the Index() action Method in a Browser

A detailed example of a controller action that returns a value of the type, `ActionResult`, will be discussed in the next section.

Invoking Controller Actions

You can define a number of action methods in a controller. To invoke an action method, you can specify a URL in the Web browser that contains the information about a controller and the action method within the controller. For example, to invoke the Registration action method in the HomeController controller of a shopping application, www.newbay.com, you need to specify the following URL in the Web browser:

http://newbay.com/Home/Registration

When the preceding URL is sent as a request to an MVC application through a Web browser, the MVC application first searches for the `HomeController` class, and then searches for the `Registration()` action method in it. If the `Registration()` action method is found, the MVC application executes the method, and then returns the response back to the browser.

Task 2.1: Creating a Controller

Passing Parameters in Controller Actions

While requesting for a Web page, you may want to specify information other than the name of the Web page. For example, while requesting for the product details, you may want to specify the category of the product. This additional information can be sent to the server in the form of a query string included in the URL. For example, consider the following URL:

http://www.demo.com/home/browse?Category=Soap

A query string is a part of a URL. It is used to pass data to Web servers. The query string is always starts with a ? character followed by the data to be passed. The data is passed in the following format in query string:
`ParameterName=Value`
Where:

ParameterName: Is the name of the variable, such as `Category`.

Value: Is the value of that parameter, such as `Soap`.

In the preceding URL, a query string is defined that sets the value, `Soap`, for the `Category` parameter. The query string passes this information to the `browse()` action method.

The `browse()` action method can access the value for the `Category` parameter by using its arguments. For example, consider the following code snippet:

```
public string browse(string Category)
 {
   return ("Category is: " + Category);
 }
```

NOTE *To access the value of a query string parameter in an action method, the parameter name of the action method must be the same as the name of the query string parameter.*

In the preceding code snippet, an action method named `browse()` is defined, which accepts a string argument named `Category`. This argument provides the `browse()` action method access to the category value passed through the query string. Once invoked, the action method, `browse()`, will display the following output:

Category is: Soap

Now, consider the following URL:

http://localhost:62053/home/browse/5

In the preceding URL, a numeric parameter, 5, with the default name, `id`, is included in the URL. However, the parameter is directly embedded within the URL, and not as a query string.

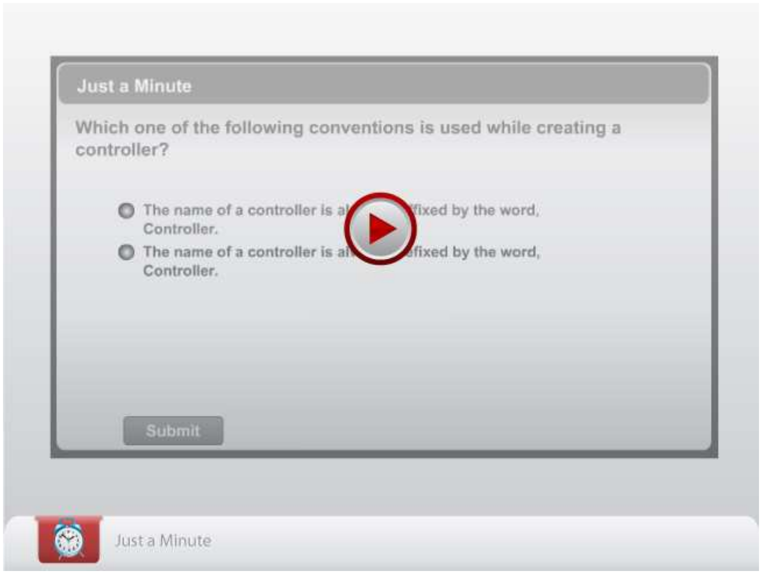
You can access the value of the ID parameter within the browse() action method by using the following code snippet:

```
public string browse(int id)
 {
   return "ID is: " + id;
}
```

In the preceding code snippet, an action method named `browse()` is defined with an `id` parameter.

Once the action method is invoked through a URL, the browser renders the view and displays the following output:

ID is: 5

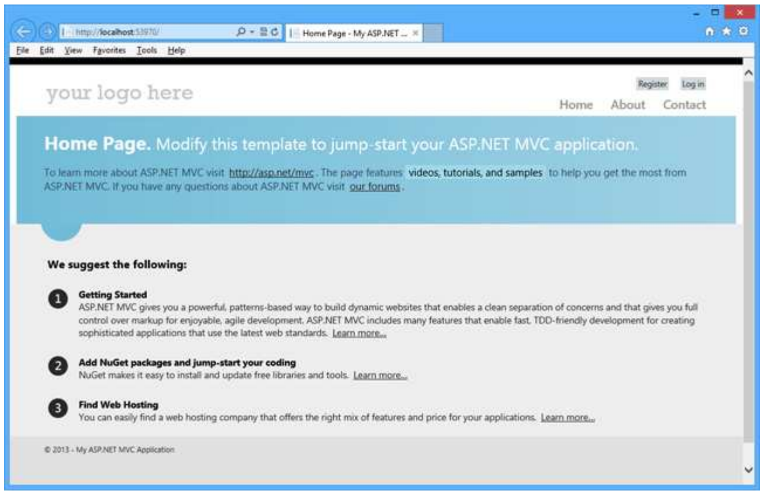

Just a Minute
Which one of the following conventions is used while creating a controller?
The name of a controller is a...fixed by the word, Controller.
The name of a controller is a...fixed by the word, Controller.
Submit
Just a Minute

Working with Views

You have learned that controller actions can return strings, which are then displayed on the browser. However, most controller actions need to display dynamic information, usually in the HTML format. To display dynamically generated HTML content, a controller action can be made to return a view instead of a string.

A view is a combination of HTML markup and code that runs on the Web server. The code is embedded in the Web pages using the Razor syntax. A Web application contains multiple views, where each view is responsible for either displaying some information to users or taking inputs from a user.

The following figure represents a view displaying some information to the users.



Home Page. Modify this template to jump-start your ASP.NET MVC application.

To learn more about ASP.NET MVC visit http://asp.net/mvc . The page features videos, tutorials, and samples to help you get the most from ASP.NET MVC. If you have any questions about ASP.NET MVC visit our forums .

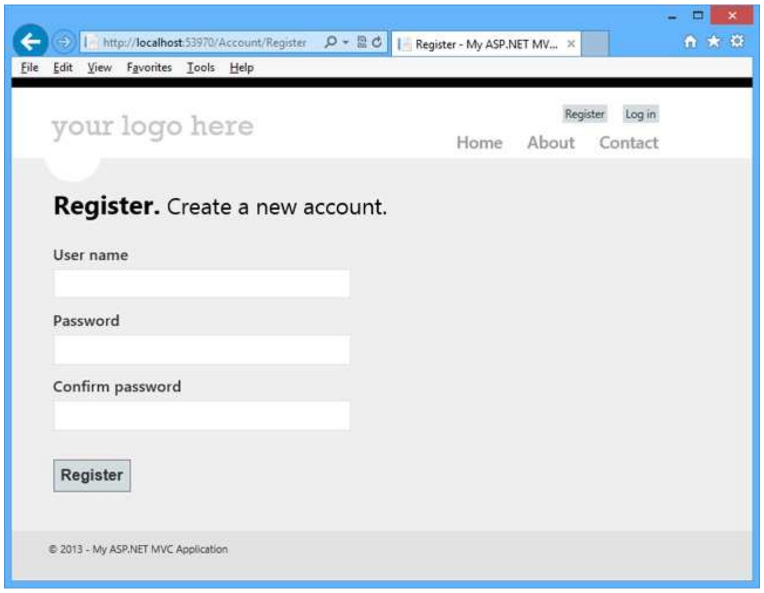
We suggest the following:

1. **Getting Started**
   ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and that gives you full control over markup for enjoyable, agile development. ASP.NET MVC includes many features that enable fast, TDD-friendly development for creating sophisticated applications that use the latest web standards. Learn more...
2. **Add NuGet packages and jump-start your coding**
   NuGet makes it easy to install and update free libraries and tools. Learn more...
3. **Find Web Hosting**
   You can easily find a web hosting company that offers the right mix of features and price for your applications. Learn more...

© 2013 - My ASP.NET MVC Application

A View Displaying Some Information to the Users

The following figure represents a view that takes the input from a user.


http://localhost:53970/Account/Register
Register - My ASP.NET MV...
File Edit View Favorites Tools Help
your logo here
Register Log in
Home About Contact

Register. Create a new account.

User name

Password

Confirm password

Register

© 2013 - My ASP.NET MVC Application

A View that Takes Input from a User

Specifying the View for an Action

You have already seen that user requests are processed by the action methods defined within the controllers. Once a request is processed, a controller renders an appropriate view that is shown to the user. For this, an action method needs to return a view. Consider the following code snippet:

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

As shown in the preceding code snippet, a view is returned through `View()` method from within the `Index()` action method of the Home controller. When a view is returned without specifying any name for the view in the `View()` method, MVC searches for the view with the same name as the action method within the /Views/<ControllerName> folder. By convention, the Views folder contains one folder for every controller, with the same name as the controller. Therefore, in the preceding example, a view with the name Index, which is created inside the Views/Home folder, will be rendered.

You have seen that the default convention to locate a view is /Views/<ControllerName>/<ActionMethodName>. However, you can also render a different view from within an action method by specifying the name of the view as a parameter of the `View()` method, as shown in the following code snippet:

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

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

You can also render a view defined in a different folder other than the default folder created by MVC for every controller. To render a view named `Index` defined in the /Views/Special folder, you need to specify its full path, as shown in the following code snippet:

```
public class HomeController :
Controller
 {
   public ActionResult Index()
     {
      return View("~/Views/Special/
Index.cshtml");
     }
 }
```

The preceding code snippet will render the view named Index defined inside the /**Views**/**Special** folder.

In the preceding code snippet, the tilde (~) sign has been used to refer to the root application folder. While using the tilde symbol, you must give the file extension of the view because this method bypasses MVC's default convention of finding a view.

In most cases, a view returns an instance of the `ViewResult` class, which, in turn, is derived from the `ActionResult` class.

Task 2.1: Creating a View

Activity 2.1: Creating Controllers and Views

Passing Data from a Controller to a View

A controller interacts with the model to fetch application data. However, the data is displayed to the user through a view. Therefore, there needs to be some mechanism to pass the data from a controller to a view. The following two common techniques are used to pass data from a controller to a view:

- ❑ `ViewData`
- ❑ `ViewBag`

ViewData

`ViewData` is a dictionary of objects derived from the `System.Web.Mvc.ViewDataDictionary` class. It enables you to pass the data between a controller and a view. Here, values can be set using key/value pairs in the controller action method, as shown in the following code snippet:

```
ViewData["Message"] = "Welcome to our
website";
ViewData["ServerTime"] = DateTime.Now;
```

In the preceding code snippet, two keys, `Message` and `ServerTime`, are added with values `Welcome to our website` and current system date to the `ViewData` dictionary. You can access the values added in the preceding two keys in a view by using the Razor syntax, as shown in the following code snippet:

```
<p>
The message is: @ViewData[“Message”]
The Date and Time is: @ViewData
[“ServerTime”]
</p>
```

ViewBag

`ViewBag` is a dynamic object that allows passing data between a controller and a view. You can add properties of multiple types and their values by using the `ViewBag` object in a controller action method. Further, in the view, you can use the `ViewBag` object to access the value added in the action method. You can add properties in a `ViewBag` object using a very simple syntax, as shown in the following code snippet:

```
ViewBag.Message = “Welcome to our
website”;
ViewBag.ServerTime = DateTime.Now;
```

In the preceding code snippet, two properties are added in the `ViewBag` object. The first property, `Message`, adds a string to it, and the second property, `ServerTime`, sets the current date and time to it. You can access these two properties in a view by using the Razor syntax, as shown in the following code snippet:

```
<p>
The message is: @ViewBag.Message
The Date and Time is:
@ViewBag.ServerTime
</p>
```

The preceding code snippet will retrieve the message and the current date and time from the `ViewBag` object method and render them in the view. `ViewBag` is just a dynamic wrapper over the `ViewData` dictionary. It means that a value saved in the controller action by using `ViewBag` can also be accessed by using `ViewData` in a view. For example, the Message property set by using `ViewBag` in the action method can be accessed by using `ViewBag` and `ViewData` in the following ways in a view:

```
@ViewBag.Message
@ViewData[“Message”]
```

One of the most common differences between `ViewBag` and `ViewData` is that `ViewBag` works only in those cases where the key being accessed is a valid C# identifier. For example, if you have used `ViewData[“Customer name”]` and set its value, then you cannot access this value by using `ViewBag`. This is because a key with space is not a valid C# identifier and the code will not run.

The Razor View Engine

To render a view inside a browser, MVC uses Razor as the default view engine. The Razor view engine uses a specific syntax to manage rendering of Web pages. Razor is a clean, lightweight, and simple view engine with a minimum amount of syntax and code. Razor allows embedding server-side code in a view.

Apart from the Razor view engine MVC 4 also supports the ASPX view engine. Prior to MVC 3, ASPX was the only supported view engine. Razor was introduced with MVC 3 and is the default view engine moving forward.

Defining Razor

Razor is a markup syntax that allows you to embed server-side code (written in C# or VB) in an HTML markup. A file containing server-side code in the C# syntax has an extension of .cshtml. However, a file containing server-side code in VB has an extension of .vbhtml.

The Razor Syntax Rules

The Razor syntax follows certain rules. Some of these are:

- ❑ The @ character indicates a transition from markup to code.
- ❑ Razor code statements end with semicolon (;).
- ❑ The C# code is case-sensitive.
- ❑ Variables should be declared by using the `var` keyword.
- ❑ Strings should be enclosed within quotation marks.

You have seen some of the basic Razor syntax rules. Now, let us discuss the syntax of various constructs in Razor.

The Razor Syntax Guide

The Razor view engine is responsible for interpreting the server-side code embedded inside a view file. For this, Razor needs to distinguish the server-side code from the markup code. To distinguish the server-side code from the markup code, Razor uses the @ symbol.

Including Code Expressions

Consider the following code snippet:

```
<h1>Displaying @myList.length items.</
h1>
```

In the preceding code snippet, `@myList.length` is interpreted as the server-side code. Notice that here, you do not need to specify the end of the Razor code. Razor is smart enough to distinguish between the markup code and the server-side code. However, sometimes, you may need to override the logic that Razor uses to distinguish the server-side code. For example, if you want to display the @ symbol within an email address, you can use @@, as shown in the following code snippet:

```
<h1>your email ID is: mark@@newbay.com
</h1>
```

In the preceding code snippet, Razor interprets the first `@` symbol as an escape sequence character. Razor uses an implicit code expression to determine parts of a line that are server-side code. However, sometimes, Razor interprets an expression as markup that needs to be run as server-side code. For example, consider the following code snippet:

```
<span>Price of product including Sales
Tax: @Model.Price * 1.2</span>
```

The preceding code snippet renders output as `Price of product including Sales Tax: 5 * 1.2`, if the price of the product is 5. You can use parentheses to explicitly delimit expressions, as shown in the following code snippet:

```
<span>Price of product including Sales
Tax: @(Model.Price * 1.2)</span>
```

The preceding code snippet will display output as `Price of product including Sales Tax: 6`, if the price of the product is 5.

Including Code Blocks

Sometimes, you may need to write multiple lines of server-side code. You can do this without prefixing every line of code with the @ symbol by using the Razor code block. In a Razor code block, lines are enclosed with curly braces in addition to an @ symbol, as shown in the following syntax

:

```
@{
//Server-side codes
----------------------
}
```

Combining Code and Markup

You can use a Razor code block to run conditional statements and loop through collections. Here, you can mix the code blocks with the markup to generate the required output.

Razor loops are useful for displaying a list of items in a collection. For example, consider the following code snippet:

```
@foreach (var item in products)
{
<li> The item name is @item.</li>
}
```

In the preceding code snippet, the `foreach` loop is defined, which displays each item individually in the products collection. In the preceding code snippet, the `@foreach` loop automatically transitions to the markup with the opening `<li>`tag, transitions back to the code when the `@item` is encountered. Moreover, Razor automatically transitions back to the markup when the closing `<li>`tag is encountered.

In addition, you can use conditional constructs for making decisions on the basis of some condition. For example, consider the following code snippet:

```
@if (@ViewBag.Product!=null) {
<text>Product is in stock.</text>
}
```

In the preceding code snippet, the `@if construct` is defined, which displays the text, `Product is in stock`, if the condition, `@ViewBag.Product!=null`, is satisfied.

Including Comments

You can declare a Razor comment within the @* and *@ delimiters, as shown in the following code example:

```
@* You comment goes here. *@
```

Using Partial Views

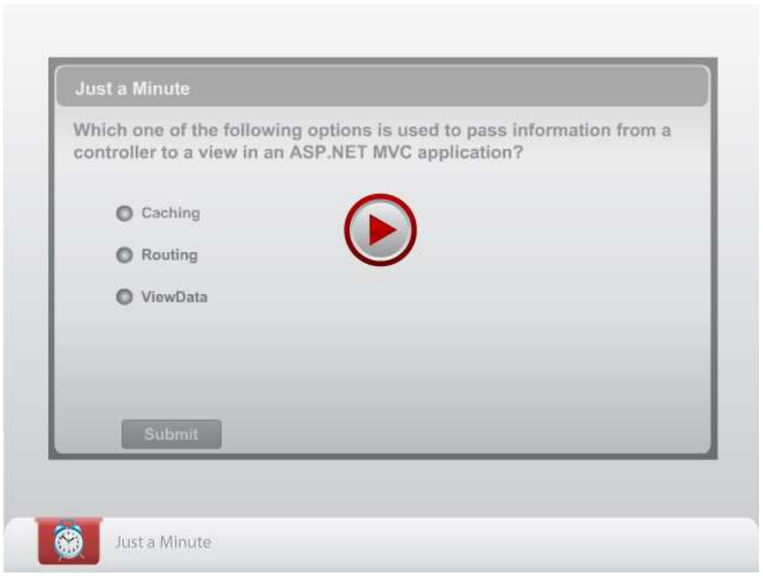
Consider a scenario where you want to include some common markup in a number of views. One option is to include the common markup in each view. However, this is a time-consuming task as you have to write the markup on every view. In addition, if you need to make a change in the markup, the same change needs to be applied on all the views that contain the markup. To overcome this problem, you can create a partial view and invoke it wherever required. A partial view is a part of a view that can be used on multiple pages. Partial views are created in the /Views/Shared folder. By convention, the name of a partial view is prefixed with an underscore (_) character.

You can render a partial view in a view by using the `@Html.Partial()` method. MVC inserts the partial view, HTML, into the parent view, and then returns the complete Web page to the browser. For example, the following code snippet renders a partial view named `_partialsummary` within a view:

```
<div>@Html.Partial("_partialsummary")
</div>
```

In the preceding code snippet, a partial view named `_partialsummary` is rendered within a `<div>` tag.

Task 2.2: Creating a Partial View



Just a Minute

Which one of the following options is used to pass information from a controller to a view in an ASP.NET MVC application?

- Caching
- Routing
- ViewData

Activity 2.2: Passing Data between Controllers and Views

Routing Requests to Controller Actions

In an ASP.NET MVC application, the controller action to be executed in response to a user request is determined by the Uniform Resource Locator (URL) of the incoming request. The URL is a widely used user interface for the Web. However, not enough attention is paid to the user-friendliness of URLs, and we often find long and complicated URLs, such as:

http://www.demo.com/product/products.aspx?category=laptop&manufacturer=dell&id=1243&color=black

Observe the preceding URL and you can understand that the website has a directory structure that contains a product folder and a **products.aspx** file within that folder. This requires a direct relation between the URL and the physical files stored on the disk. After receiving this URL request, the Web server directs the request to the **products.aspx** file, which contains the code and markup to render a response to the browser. The **products.aspx** file uses the data in the query string to identify the type of content to display.

In the preceding example, there is a one-to-one mapping between the URLs and the file system. However, this is usually not the case in an MVC application. In an MVC application, whenever a user request is generated, the URL of the incoming request is routed to a controller action. The mechanism for locating an appropriate action method for a given URL is called routing. Routing in the ASP.NET MVC framework serves the following two main purposes:

- ❑ It maps the URLs of incoming requests with controller actions.
- ❑ It builds outgoing URLs that correspond to controller actions.

In an ASP.NET MVC application, because the URLs do not need to map with files, the URLs can be more descriptive of the user's action, and do not need to use long and complicated query strings.

For example, consider the following URL:

http://www.demo.com/product/products/laptop/dell/1243/black

The preceding URL points to the `products()` action method of the product controller. The routing mechanism passes the values, laptop, dell, 1243, and black, to the action method that the URL points to. The action method can use these values to determine the content to be displayed.

A user can easily modify the preceding URL to navigate to another page of the site. In addition, one important benefit of this kind of URL is Search Engine Optimization (SEO). SEO is the process of optimizing a URL for a search engine to improve the page ranking in search engine results. SEO reflects what people search for by using the keywords typed into search engine and how search engines work to process these requests. Search engines mainly work using URLs. Specifying a meaningful and more comprehensive URL is critical to make the application more search engine-friendly.

An ASP.NET MVC application defines the default route for mapping URLs in a given pattern with specific controller actions. However, you can define additional routes if there is a need to provide custom routes.

The Default Route

Every MVC application needs at least one route to define how the application should handle user requests.

Whenever you create an ASP.NET MVC application, the application, by default, generates a route for routing user requests. The Global.asax file has a method, `Application_Start()`, defined in it. This is the first method that is invoked when an MVC application starts. Finally, this method invokes the `RegisterRoutes()` static method, which is defined in the **RouteConfig.cs** file in the **App_Start** folder. The `RegisterRoute()` method has the definition of the default route, as shown in the following code snippet:

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

The preceding code snippet defines a route named `Default`. This route defines a URL pattern, `{controller}/{action}/{id}`, with three segments: `controller`, `action`, and `id`. In addition, it defines the default value for each segment in the pattern. The default value of the `controller` segment is defined as `Home`, the `action` segment is `Index`, and id is an optional parameter. Therefore, if a URL does not specify an action method, the request is routed to the default action method, `Index()`, in the specified controller. Similarly, if a URL request does not specify a controller, the request is routed to the `Index()` action method of the `Home` controller.

The following table shows the requested URLs and the controllers, actions, and ID parameters that they map with, as per the default routing scheme:

URL Request	*Controller*	*Action Method*	*ID*
http://www.demo.com/	*Home*	*Index*	
http://www.demo.com/product	*product*	*Index*	
http://www.demo.com/product/browse	*product*	*browse*	
http://www.demo.com/product/browse/5	*product*	*browse*	*5*

The URL Mapping with Controller Actions as per the Default Routing Scheme

Routing Patterns

The default route defines the routing mechanism for URLs of the form, **http://<domain>/<contoller>/<action>/<id>**. However, to handle URLs that use a different form, additional routes need to be defined. To define a new route, a URL pattern needs to be created. This URL pattern defines a route pattern that a route will match with. A route pattern can contain variable placeholders or literal values or a combination of these two. These placeholders and literals are placed in segments of the URL separated by the slash (/) character. Placeholders are enclosed in braces, { and }.

For example, consider the following URL pattern and the matching URL:

`URL pattern: "{controller}/{action}/{id}"`

URL: http://www.demo.com/employee/details/2

In the preceding URL, the three placeholders, `{controller}`, `{action}`, and `{id}`, have the values, `employee`, `details`, and `2`, respectively. You can also define multiple placeholders in a segment. However, these placeholders must be separated by literal values. For example, consider the following URL:

http://www.demo.com/dell-US/show

In the preceding URL, the `{company}-{country}/{action}` pattern is used. You can see that in the first segment, two placeholders `{company}` and `{country}` are created, separated by a dash (-) literal.

A route pattern can also have a combination of literal values and variable placeholders, as shown in the following route pattern:

`blog/{action}/{entry}`

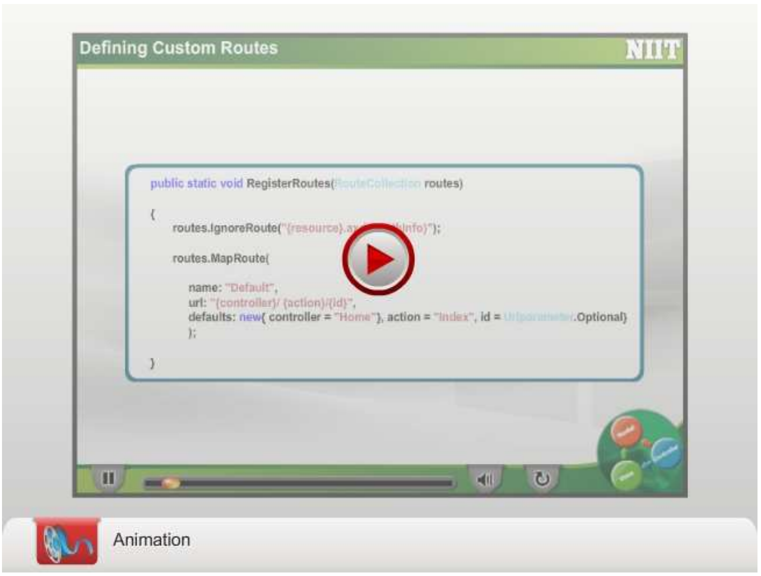
In the preceding route pattern, the first segment has a literal value, blog, and the second and third segments define variable placeholders.

For example, consider the following URL:

http://www.demo.com/blog/show/321

In the preceding URL, the `blog/{action}/{entry}` pattern is used. Here, `blog` represents a literal value. The value, `show`, is specified for the placeholder, `action`, and the value, `321`, is specified for the placeholder, `entry`.

The preceding URL is routed to the `show()` action method of the blog controller, and the value 321 is passed to the `show()` action method as a parameter.



```
public static void RegisterRoutes(RouteCollection routes)

{
    routes.IgnoreRoute("{resource}.a[illegible]Info}");

    routes.MapRoute(

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

}
```

Ordering Routes

For large applications, you may need to register multiple routes. However, when you define multiple routes, you should define them in a specific order. This is because whenever a URL request is received by the application, the route matching is done from the first route to the last route. While matching the routes, whenever a match is encountered, no more routes are tried after that. This behavior can cause unexpected problems.

For example, consider the following two routes:

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

The first route contains two placeholder segments 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 constant segment, `Admin`, and a variable placeholder segment, and sets the default value of the controller parameter to `Admin` and the action parameter to `Create`. It causes problems in routing. This is because the first route can match just about any value entered for a controller and action segment. This would even include the value, `Admin` for the controller segment. It means that when you call a URL by using the value, `Admin`, for the first segment, the first route will be called and the second route will never be called.

To fix this problem, you need to switch the order of the routes so that the second route named admin is defined before the first route named general.

URL Routing Constraints

While creating routes, you may need to place extra constraints to ensure that the route matches only with the appropriate requests. Providing constraints along with the route pattern gives a better control over how and when a route matches an incoming request URL. For example, if you want a URL in the form **toys/45** to specify a toy with the `id` **45**, you create a route pattern like `toys/{id}`. However, this route pattern can also match with the URL **toys/create**. For `id`, you must constraint the URL to match only with the segments comprising digits. This is where constraints are useful along with the route pattern.

Constraints allow you to apply a regular expression to a route segment to ensure that only appropriate request will match. For example, consider the following route:

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

The preceding route has three segments: `controller`, `action`, and `id`. In addition, the default value for the `controller` segment is set to `Home`, the `action` segment is set to `Index`, and id is an optional parameter. In the preceding route, a constraint is applied on the `id` segment, which specifies that the id parameter can match only with an integer value. The following table gives an example of URLs that may or may not match the preceding route.

URL	*Match?*
http://www.demo.com/product/toy/2341	*Yes*
http://www.demo.com/product/toy/car	*No*
http://www.demo.com/product/toy	*Yes*
http://www.demo.com/product/	*Yes*
http://www.demo.com/	*Yes*

The URLs and their Match Result with a Route

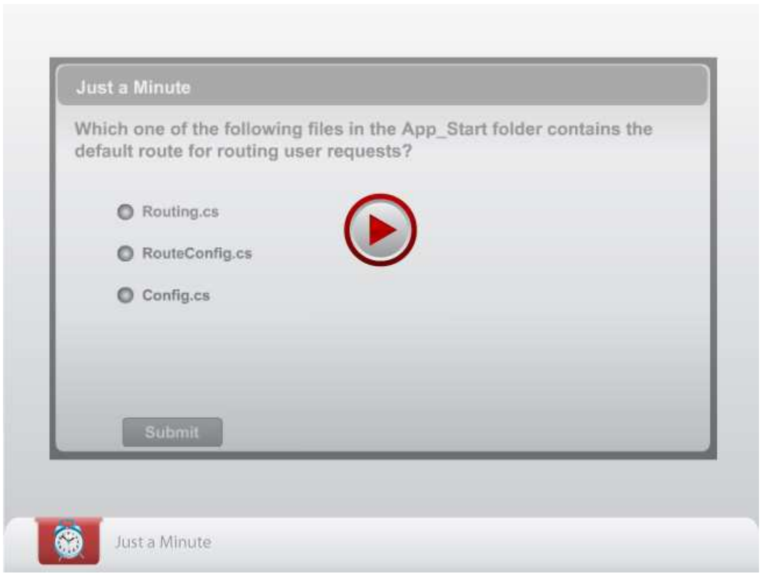
Ignoring a Route

In contrast to defining routes that map with controllers and actions, you can prevent URLs from being evaluated against routes. For this, you can use the `IgnoreRoute()` method that adds a special route and tells the routing engine to ignore requests for any URL that matches the given pattern.

For example, consider the following code snippet:

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

The preceding code snippet ignores any request for files with the extension, .axd. This file extension is used for common ASP.NET handlers, such as **Trace.axd** and **WebResource.axd**. With the preceding call to the `IgnoreRoute()` method, the requests for **.axd** files are handled as normal requests to ASP.NET and are not handled by the routing engine.



Just a Minute

Which one of the following files in the App_Start folder contains the default route for routing user requests?

- Routing.cs
- RouteConfig.cs
- Config.cs

Summary

In this session, you learned that:

- Controllers are implemented in an MVC application as C# classes inherited from .NET Framework’s built-in `Controller` class.
- A controller class contains an application logic in the form of various public methods called action methods.
- The controllers are placed in a folder named **Controllers** under the Web application folder.
- To invoke an action method, you can specify a URL in the Web browser that contains the information about a controller and the action method within the controller.
- A view is a combination of HTML markup and code that runs on the Web server.
  The following two common techniques are used to pass data from a controller to a view:
  - `ViewData`
  - `ViewBag`
- `ViewData` is a dictionary of objects derived from the `ViewDataDictionary` class.
- `ViewBag` is just a dynamic wrapper over the `ViewData` dictionary.
- Razor is a markup syntax that allows you to embed server-side code (written in C# or VB) in an HTML markup.
- The Razor view engine is responsible for interpreting the server-side code embedded inside a view file.
- A partial view is a part of a view that can be used on multiple pages.
- You can render a partial view in a view by using the `@Html.Partial()` method.
- The mechanism for locating an appropriate action method for a given URL is called routing.
- SEO is the process of optimizing a URL for a search engine to improve the page ranking in search engine results.
- An ASP.NET MVC application defines the default route for mapping URLs in a given pattern with specific controller actions.
- Every MVC application needs at least one route to define how the application should handle user requests.
- The default route defines the routing mechanism for URLs of the form, **http://<domain>/<contoller>/<action>**.
- A route pattern can contain variable placeholders or literal values or a combination of these two.
- Providing constraints along with the route pattern gives a better control over how and when a route matches an incoming request URL.

Chapter 3

Working with Models and Helper Methods

Most applications have some data associated with them. Therefore, interaction with the data becomes critical for any application. In an MVC application, the component that enables interaction with data is called model. To access and retrieve data in an application, you need to know how to work with models.

A user interacts with an application with the help of views. Therefore, it is important to design effective views. ASP.NET MVC provides a number of helper methods that help you create views easily and provide several additional benefits.

This chapter discusses how to work with models and helper methods in an ASP.NET MVC application.

Objectives

In this chapter, you will learn to:

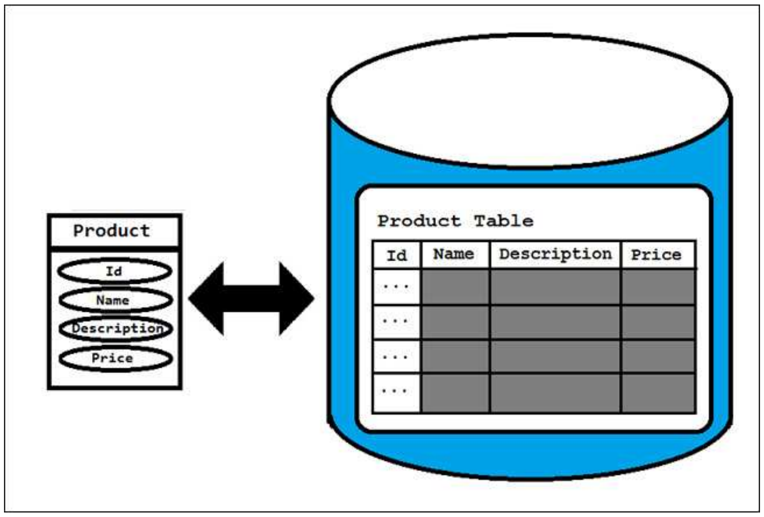
- ❑ Work with Models
- ❑ Work with HTML and URL Helpers

Working with Models

Consider the Web application of Newbay shopping store. The Project Head, Callie Jones, instructs the team to implement an additional functionality that allows customers to view the product categories and the products available in the selected category online. In addition, Callie wants that the application should allow the administrator to add details of products. Further, she wants that the application should allow customers to place online orders.

The software development team has already created the Home, About Us, and Contact Us pages of the application. Now, they need to manage the data related to the products. In an MVC application, this is the model that manages all the tasks related to handling data. The model represents the data and the business logic of an application.

A model consists of a set of classes, where the objects of the classes represent the data associated with the application. Suppose you have the data of products, which includes product name, description, and price. To represent this data, you need to create a model class named `Product` with properties, such as `Id`, `name`, `description`, and `price`. The structure of this class should be similar to the structure of the `Product` table in the database in such a way that there is a direct mapping between the properties of the `Product` class and the fields of the `Product` table. This is called object-relational mapping (ORM). The following figure shows a direct mapping between the properties of the `Product` class and the fields of the `Product` table.



The Object-Relational Mapping

A model also represents the business logic of an application. For example, in case of the Newbay website, the developer has implemented the functionality that allows the administrator to compute the total monthly sales generated online. In this case, the model is responsible for performing the calculation.

Once defined, you can access a model within a controller. Moreover, you can also pass model data from a controller to a view. This view can be created manually or automatically. The ASP.NET MVC framework provides a feature called scaffolding, using which you can create auto-generated controllers and views.

Let us learn how to perform the preceding tasks individually.

You will learn more about scaffolding later in this chapter.

Creating a Model

Consider the scenario of Newbay where the application needs to maintain information about the products, such as Id, product name, description, and price. For this, the team needs to add a class in the model that defines the required properties of the product.

The following code snippet declares the model class:

```
public class Product
{
public long Id {get; set;}
public string name {get; set;}
public string description {get; set;}
public int price {get; set;}
}
```

The preceding code snippet creates a model class named `Product` containing properties, such as `Id`, `name`, `description`, and `price`. As a convention, all model classes are stored in the `Models` folder within an ASP.NET MVC application.

Task 3.1: Creating a Model

Accessing a Model Within a Controller

Consider the scenario of the Newbay application, where a user requests for some information related to a product through the following URL:

www.Newbay.com/Home/Product

Once received, the server sends the request to the `Product` action method of the `Home` controller. The `Product` action method coordinates with the model to access the data and displays it through a view.

To access the model from the `Product` action method, you need to create an object of the model class in the `Product` action method, as shown in the following code snippet:

```
public ActionResult Product()
{
 var product = new
 Newbay.Models.Product();
 product.name = "Samsung 5676 Mobile";
 product.description = "Newly launched
 with touch screen";
 product.price = 15000;
 return View();
}
```

In the preceding code snippet, `product` is an object of the `Product` class. The data related to the products, such as `name`, `description`, and `price`, is defined statically. To refer to the `Product` model class, the qualifier, `Newbay.Models`, is used. This qualifier can be omitted by including the namespace, `Newbay.Models`, at the top of the controller class, as shown in the following code snippet:

```
using Newbay.Models
```

Here, `Newbay.Models` is the namespace that contains the `Product` model class, and `Newbay` is the name of your application.

Passing Model Data from a Controller to a View

You have seen how to access a model within a controller. However, to be displayed to a user, the model data needs to be made accessible to a view. For this, the controller can pass the model data to the view at the time of invoking the view. The model data passed could be a single object or a collection of objects.

Passing a Single Object

Consider a controller action that can be invoked to retrieve the details of a product. To display the details to a user, you need to create a view that can be used to display the value of a single product. In addition, you need to pass the data associated with the product to the view.

You can pass the model data from a controller to a view by using `ViewBag`. To pass the data, you need to add the highlighted portion of the following code snippet in the `Product` action method of the `Home` controller:

```
public ActionResult Product()
 {
 var product = new Product();
 product.name = "Samsung 5676 Mobile";
 product.description = "Newly launched
 with touch screen";
 product.price = 15000;
 ViewBag.product = product;
 return View();
 }
```

In the preceding code snippet, an instance of the `Product` model class is passed to a view by using `ViewBag`. To access the data stored in `ViewBag` from within the view, you need to add the following code snippet in the view:

```
@ViewBag.product.name<br/>
@ViewBag.product.description<br/>
```

```
@ViewBag.product.price<br/>
```
In the preceding code snippet, the razor syntax is used to display the property values of the object passed through `ViewBag`.

You can also use the following code snippet as an alternative to the previous code snippet:
```
@{
 var product = ViewBag.product;
}
@product.name <br/>
@product.description <br/>
@product.price <br/>
```
In the preceding code snippet, a reference of the object passed through `ViewBag` is stored in a variable, `product`. The properties of the product are then displayed using this object.

As an alternative to using `ViewBag`, you can pass the data to a view by passing the object as a parameter to the view, as shown in the highlighted portion of the following code snippet:
```
public ActionResult Product()
 {
 var product = new Product();
 product.name = "Samsung 5676 Mobile";
 product.description = "Newly launched
 with touch screen";
 product.price = 15000;
 return View(product);
 }
```
To access the passed information within the view, you need to add the following code snippet in the view:
```
@Model.name <br/>
@Model.description <br/>
@Model.price <br/>
```
Alternatively, you can also use the following code snippet to access the passed information in the view:
```
@{
 var product = Model;
}
@product.name <br/>
@product.description <br/>
@product.price <br/>
```

Passing a Collection of Objects

Consider a scenario where you need to display the details of multiple products to a user. For this, you need to pass the data associated with all the products to the view. You can use the `ViewBag` technique to pass multiple values from a controller to a view. To pass the data, you need to add the following code snippet in the `Productlist` action method of the `Home` controller:
```
public ActionResult Productlist()
 {
 var products = new List<Product>();
 var product1 = new Product();
 product1.name = "Samsung 5676 Mobile";
 product1.description = "Newly launched
 with touch screen";
 product1.price = 15000;
 var product2 = new Product();
 product2.name = "Dell Inspiron 1545
 laptop";
 product2.description = "500 GB HDD
 with 6 GB RAM";
 product2.price = 40000;
 var product3 = new Product();
 product3.name = "Sony Bravia LED tv";
 product3.description = "40 inch Full
 HD";
 product3.price = 30000;
 products.Add(product1);
 products.Add(product2);
 products.Add(product3);
 ViewBag.products = products;
 return View();
 }
```
In the preceding code snippet, three instances of the `Product` model class for three different products are passed as a collection to a view by using `ViewBag`.

To access the data stored in `ViewBag` from within the view, you need to add the following code snippet in the view:
```
@{
 var products = ViewBag.products;
 }
 @foreach (var p in products)
 {
 @p.name<br/>
 @p.description<br/>
 @p.price<br/>
 <br/>
 }
```
As an alternative to using `ViewBag`, you can pass the data to a view by passing a collection of objects as a parameter to the view, as shown in the following code snippet:
```
public ActionResult Productlist()
 {
 var products = new List<Product>();
 var product1 = new Product();
 product1.name = "Samsung 5676 Mobile";
 product1.description = "Newly launched
 with touch screen";
 product1.price = 15000;
 var product2 = new Product();
 product2.name = "Dell Inspiron 1545
 laptop";
 product2.description = "500 GB HDD
 with 6 GB RAM";
 product2.price = 40000;
 var product3 = new Product();
 product3.name = "Sony Bravia LED tv";
 product3.description = "40 inch Full
 HD";
 product3.price = 30000;
 products.Add(product1);
 products.Add(product2);
 products.Add(product3);
 return View(products);
 }
```

To access the passed information within the view, you need to add the following code snippet in the view:

```
@{
var products = Model;
}
@foreach(var p in products)
{
@p.name <br/>
@p.description <br/>
@p.price <br/>
<br/>
}
```

Task 3.2: Passing Model Data from a Controller to a View

Using Strong Typing

You have seen how to pass model data from a controller to a view. However, when the data is passed in this manner, the view is not aware of the exact type of the data being passed. As a result, you do not get the benefit of intellisense while trying to access the properties of the passed object. In addition, you do not get the benefits of strong typing and compile-time checking of code, such as correctly typed property and method names.

To secure the preceding benefits, you can typecast the model data to a specific type, as shown in the following code snippet:

```
@{
var product = Model as
Newbay.Models.Product;
}
@product.name <br/>
@product.description <br/>
@product.price<br/>
```

In the preceding code snippet, the `Model` object is cast to the type, `Newbay.Models.Product.` As a result of this casting, the product object is created as an object of the type, `Newbay.Models.Product`, and enables compile-time checking of code.

However, if you want to avoid explicit type casting of the model object, you can create a strongly typed view and obtain the benefits of strong typing. A strongly typed view specifies the type of model it expects by using the `@model` declaration, as shown in the following code snippet:

```
@model Newbay.Models.Product
```

Now, you can access the properties of the model object by using the following code snippet:

```
@Model.name <br/>
@Model.description <br/>
@Model.price <br/>
```

To avoid specifying a fully qualified type name for the model, you can use the `@using` declaration, as shown in the following code snippet:

```
@using Newbay.Models
@model Product
```

A better approach for namespaces that can be used often within views is to declare the namespace in the **web.config** file within the **Views** directory, as shown in the highlighted part of the following code snippet:

```
<namespaces>
<add namespace="System.Web.Mvc" />
<add namespace="System.Web.Mvc.Ajax" />
<add namespace="System.Web.Mvc.Html" />
<add namespace="System.Web.Routing" />
<add namespace="Newbay.Models" />
</namespaces>
```

So far you have seen how to implement strong typing while passing a single object to a view. However, if you want to pass a collection of objects, you need to use the `@model` declaration, as shown in the following code snippet:

```
@model
IEnumerable<Newbay.Models.Product>
```

The preceding declaration specifies that the view can accept a collection of objects of the type, `Product`. Now, you can access the collection of model objects in the view by using the following code snippet:

```
@{
var products = Model;
}
@foreach(var p in products)
{
@p.name <br/>
@p.description <br/>
@p.price <br/>
<br/>
}
```

Introduction to Scaffolding

So far you have manually created the controllers and views to handle the information related to a product and display the same to the user. This requires a lot of time and manual coding. To overcome this problem, ASP.NET provides a feature called scaffolding that builds an application quickly.

Scaffolding allows you to create auto-generated controllers and their corresponding views after examining the properties of the associated model. Scaffolding uses a predefined convention for naming controllers and views. In addition, it knows where to place auto-generated codes for the application to work.

Scaffolding provides various templates for creating controllers and associated views. Some of these templates are:

- ❑ **Empty MVC controller:** The empty controller template adds a class derived from the `Controller` class to the Controllers folder. This class has only one action method by the name, Index, with no code inside it.
- ❑ **MVC controller with empty read/write actions:** This template adds a controller class with action methods, such as Index, Details, Create, Edit, and Delete. These action methods have some code written inside them. However, they do not perform any useful function. You can make them functional by adding your own code and creating the views for each action method.
- ❑ **API controller with empty read/write actions:** This template adds a class derived from the `ApiController` base class. You can use this template to build a Web API for your application.
- ❑ **MVC controller with read/write actions and views using Entity Framework:** This template adds a controller class with action methods, such as Index, Details, Create, Edit, and Delete. It also generates all the required views and the code to retrieve information from a database.

In addition, scaffolding provides the following templates for creating views:

- ❑ **List:** This template generates the markup to display the list of model objects.
- ❑ **Create:** This template generates the markup to add a new object to the list.
- ❑ **Edit:** This template generates the markup to edit an existing model object.
- ❑ **Details:** This template generates the markup to show the details of an existing model object.
- ❑ **Delete:** This template generates the markup to delete an existing model object.

NOTE *Scaffolding templates gives you the basic code required to implement the create, read, update, and delete (CRUD) functionality in an application. However, you may need to customize the code/markup as per your specific requirements. Using scaffolding is not necessary. However, it can speed up the development process. If you do not like the scaffolding behavior, you can write the code/markup yourself from scratch.*

Task 3.3: Creating Controllers Using Scaffolding Templates

The List Template

The List template is used to create a view that displays a list of model objects. This list is passed to the view through a controller action, as shown in the following code snippet:

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

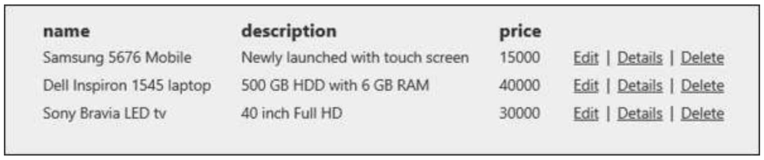
When you create a view for the `Product` model created earlier using the List template, the following markup is generated:

```
@model
IEnumerable<Newbay.Models.Product>
 @{
 ViewBag.Title = "Index";
 }
<table>
 <tr>
 <th>@Html.DisplayNameFor(model =>
 model.name)</th>
 <th>@Html.DisplayNameFor(model = >
 model.description)</th>
 <th>@Html.DisplayNameFor(model =>
 model.price)</th>
 </tr>
@foreach (var item in Model) {
 <tr>
 <td>@Html.DisplayFor(modelItem =>
 item.name)</td>
 <td>@Html.DisplayFor(modelItem =>
 item.description)</td>
 <td>@Html.DisplayFor(modelItem =>
 item.price)</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 the preceding markup, the `Html.DisplayNameFor()` helper method is used to display the names of model properties, and the `Html.DisplayFor()` helper method is used to display the values of the model properties. In addition, the `Html.Actionlink()` helper method is used to create links to edit, delete, and view details for a product. The`@model IEnumerable<Newbay.Models.Product>` method is used to type the view to a collection of `Product` objects.`@foreach (var item in Model)` is used to iterate over the collection of products and display the property values associated with each product 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.

You will learn more about helper methods in the next section.

The sample output generated by the preceding code snippet is displayed in the following figure.



name	description	price	
Samsung 5676 Mobile	Newly launched with touch screen	15000	Edit | Details | Delete
Dell Inspiron 1545 laptop	500 GB HDD with 6 GB RAM	40000	Edit | Details | Delete
Sony Bravia LED tv	40 inch Full HD	30000	Edit | Details | Delete

The View Generated by Using the List Template

This is only a sample output. The actual output will depend on the data supplied to the view by the controller.

The Create Template

The Create template can be used to generate a view that needs to be used for accepting the details of a new object to be stored in a data store.

To render a view based on the Create template, you need to create an action method. Consider an action method name `Create()` in the `Home` controller, as shown in the following code snippet:

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

The preceding code snippet will render a view named Create stored in the Views/Home folder within an MVC project. The `Create()` action method is invoked when a user clicks the Create link on the view generated through the List template.

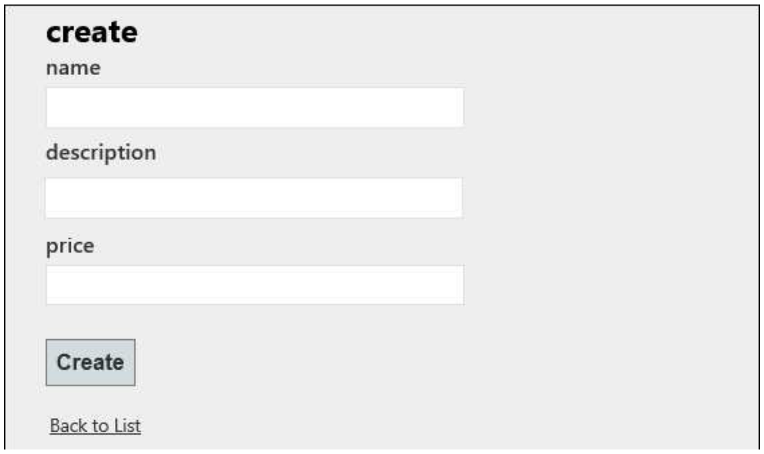
When you create a view named `Create` for the `Product` model using the Create template, the following markup is generated:

```
@model Newbay.Models.Product
@{
 ViewBag.Title = "create";
}
<h2>create</h2>
@using (Html.BeginForm()) {
@Html.ValidationSummary(true)
<fieldset>
<legend>Product</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.description)
</div>
<div class="editor-field">
@Html.EditorFor(model =>
model.description)
@Html.ValidationMessageFor(model =>
model.description)
</div>
<div class="editor-label">
@Html.LabelFor(model => model.price)
</div>
<div class="editor-field">
@Html.EditorFor(model => model.price)
@Html.ValidationMessageFor(model =>
model.price)
</div>
<p>
<input type="submit" value="Create" />
</p>
</fieldset>
}
</div>
 @Html.ActionLink("Back to List",
 "Index")
</div>
@section Scripts {
 @Scripts.Render("~/bundles/jqueryval")
 }
```

In the preceding markup, the `BeginForm` helper method marks the start of a form. The `Html.ValidationSummary()` validation helper is used to display the summary of all the error messages at one place. Further, the `Html.LabelFor()` helper method renders an HTML label element with the name of the property, and the `Html.EditorFor()` helper method displays a textbox to accept the value of a model property. In addition, the `Html.ValidationMessageFor()` validation helper is used to display a validation error message for the associated model property.

You will learn more about validation helpers in a subsequent chapter.

The preceding code snippet will generate the interface as shown in the following figure.



The View Generated by Using the Create Template

When the user fills data in the preceding form and clicks the **Create** button, an HTTP POST request is sent to the `Create()` action method within the Home controller. The data filled by the user in the form is also sent along with the request.

We already have an action method named `Create` in the Home controller. However, this action method is simply designed to render the `Create` view. To handle the HTTP POST request, another action method named `Create` needs to be created. This new action method needs to be differentiated from the previous `Create()` method by decorating it with an `HttpPost` attribute, as shown in the following code snippet:

```
[HttpPost]
public    ActionResult    Create(Product
product)
 {
 /*Code to save the details of the
 product object to a data store and
 return an appropriate view. */
 }
```

Similarly, the other action method can be decorated with an `HttpGet` attribute, as shown in the following code snippet:

```
[HttpGet]
public ActionResult Create()
{
return View();
}
```

Whenever the Create action method is requested through a URL request, the method with the `[HttpGet]` attribute is invoked. However, whenever the Create action method is invoked as a result of submitting the Create form, the method with the `[HttpPost]` attribute is invoked.

The Edit Template

The Edit template can be used to generate a view that needs to be used for modifying the details of an existing object stored in a data store.

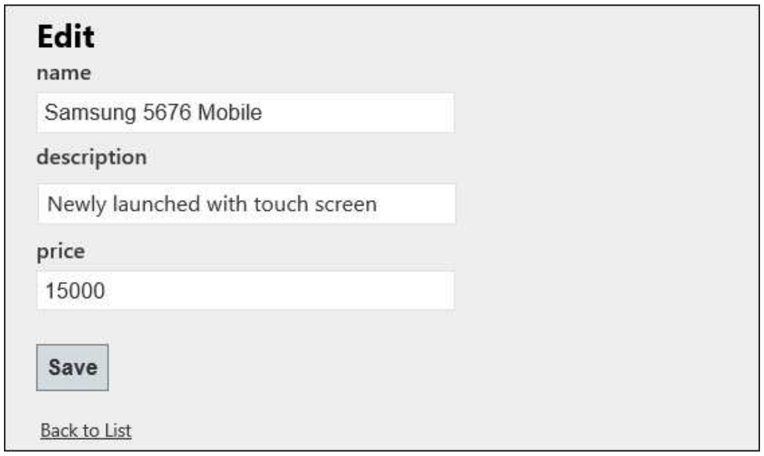
To render a view based on the Edit template, you need to create an action method that passes the model object to be edited to the view.

When you create a view named `Edit` for the `Product` model using the Edit template, the following markup is generated:

```
@model Newbay.Models.Product
@{
 ViewBag.Title = "Edit";
}
<h2>Edit</h2>
@using (Html.BeginForm()) {
 @Html.ValidationSummary(true)
 <fieldset>
 <legend>Product</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.description)
 </div>
 <div class="editor-field">
 @Html.EditorFor(model =>
 model.description)
 @Html.ValidationMessageFor(model =>
 model.description)
 </div>
<div class="editor-label">
@Html.LabelFor(model => model.price)
</div>
<div class="editor-field">
@Html.EditorFor(model => model.price)
@Html.ValidationMessageFor(model =>
model.price)
</div>
<p>
<input type="submit" value="Save" />
</p>
</fieldset>
}
<div>
@Html.ActionLink("Back to List",
"Index")
</div>
@section Scripts {
@Scripts.Render("~/bundles/jqueryval")
}
```

In the preceding markup, the `Html.LabelFor()` helper method renders an HTML label element with the name of the property. Further, the `Html.EditorFor()` helper method is used to display a textbox to accept the value of a model property. In addition, the `Html.ValidationMessageFor()` helper method is used to display a validation error message.

The preceding code snippet will generate the interface as shown in the following figure.



The View Generated by Using the Edit Template

The preceding Edit view is rendered by the `Edit()` action method in the Home controller. However, when the user edits the data in the preceding form and clicks the **Save** button, an HTTP POST request is sent to the `Edit()` action method. To handle the HTTP POST request, another action method named `Edit()` needs to be created with an `HttpPost` attribute.

The Details Template

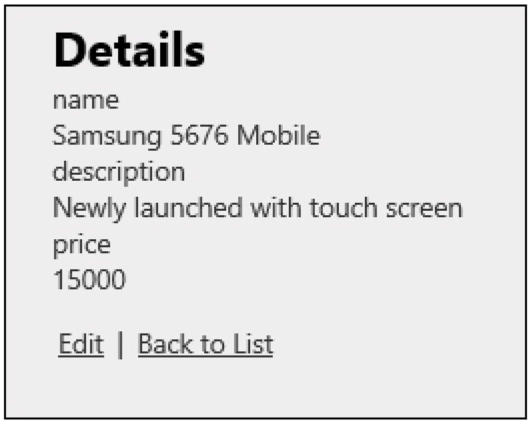
The Details template is used to create a view that displays details of a model object. The details of the model object are passed to the view through a controller action.

When you create a view for the `Product` model using the Details template, the following markup is generated:

```
@model Newbay.Models.Product
@{
ViewBag.Title = "Details";
}
<h2>Details</h2>
<fieldset>
<legend>Product</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.description)
</div>
<div class="display-field">
@Html.DisplayFor(model =>
model.description)
</div>
<div class="display-label">
@Html.DisplayNameFor(model =>
model.price)
</div>
<div class="display-field">
@Html.DisplayFor(model => model.price)
</div>
</fieldset>
<p>
@Html.ActionLink("Edit", "Edit", new
{ id=Model.ID }) |
@Html.ActionLink("Back to List",
"Index")
</p>
```

In the preceding markup, the `Html.DisplayNameFor()` helper method is used to display the names of model properties. Further, the `Html.DisplayFor()` helper method is used to display the values of the model properties.

The preceding code snippet will generate the interface as shown in the following figure.



Details

name

Samsung 5676 Mobile

description

Newly launched with touch screen

price

15000

The View Generated by Using the Details Template

The Delete Template

The Delete template can be used to generate a view that needs to be used for deleting an existing object from a data store.

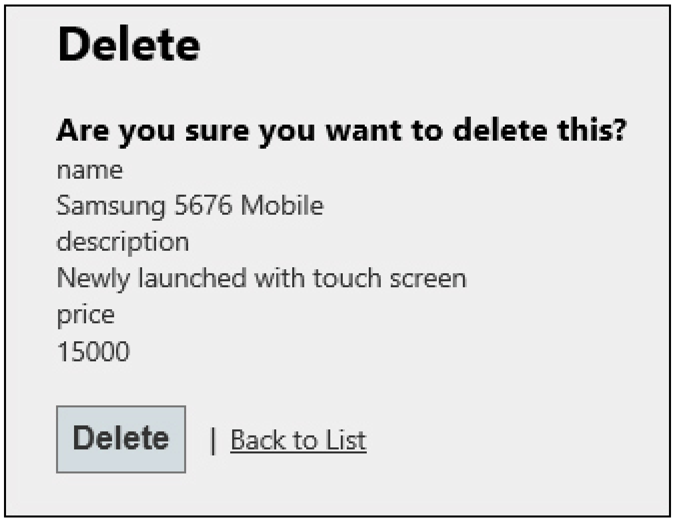
To render a view based on the Delete template, you need to create an action method that passes the model object to be deleted to the view.

When you create a view for the `Product` model using the Delete template, the following markup is generated:

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

In the preceding markup, the `Html.DisplayNameFor()` helper method is used to display the names of model properties. Further, the `Html.DisplayFor()` helper method is used to display 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.

The preceding code snippet will generate the interface as shown in the following figure.



Delete

Are you sure you want to delete this?

name

Samsung 5676 Mobile

description

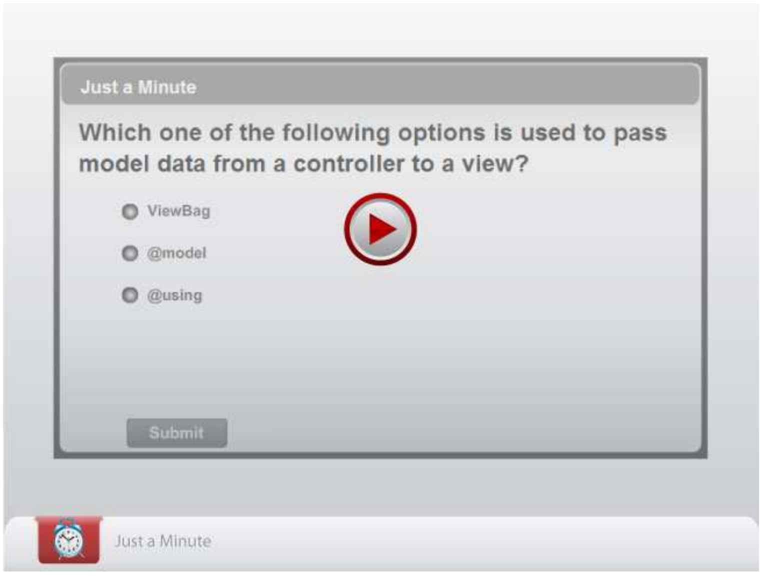
Newly launched with touch screen

price

15000

The View Generated by Using the Delete Template

The preceding Delete view is rendered by the `Delete()` action method in the Home controller. However, when the user clicks the **Delete** button, an HTTP POST request is sent to the `Delete()` action method. To handle the HTTP POST request, another action method named `DeleteConfirmed()` needs to be created with an `HttpPost` attribute.



Just a Minute

Which one of the following options is used to pass model data from a controller to a view?

- ViewBag
- @model
- @using

Working with HTML and URL Helpers

Consider a scenario where the development team working on the Newbay shopping application needs to give an option to the user to access the details of different products online. Moreover, the user should be able to navigate the different Web pages easily. In order to create the views for these features, you can use the HTML markup.

Although developing views using HTML is quite simple, yet it often becomes difficult to handle specific issues using HTML. Some of these issues are:

- Ensuring that a URL within a link actually points to the correct location
- Ensuring that form elements have proper names and are bound to the correct values to implement model binding

To overcome these problems, ASP.NET MVC offers a standard set of helper methods. These helpers help you work with HTML and URLs. The benefit of using these helpers is that they help you generate links and URLs from the routing configuration. As a result, a change in routes is automatically reflected in the links and URLs.

Some of the helper methods that can be used in building an MVC application are:

- `Html.ActionLink()`
- `Html.BeginForm()` and `Html.EndForm()`
- `Html.Label()` and `Html.LabelFor()`
- `Html.DisplayNameFor()` and `Html.DisplayFor()`
- `Html.TextBox()` and `Html.TextBoxFor()`
- `Html.TextArea()` and `Html.TextAreaFor()`
- `Html.EditorFor()`
- `Html.Password()` and `Html.PasswordFor()`
- `Html.HiddenFor()`
- `Html.CheckBox()` and `Html.CheckBoxFor()`
- `Html.DropDownList()` and `Html.DropDownListFor()`
- `Html.RadioButton()` and `Html.RadioButtonFor()`
- `Url.Action()`

Html.ActionLink()

The `Html.ActionLink()` method generates a hyperlink (anchor tag) that points to a controller's action method. The `Html.ActionLink()` helper uses the internal routing mechanism to generate the target URL for the hyperlink. The `Html.ActionLink()` helper method has the following syntax:

```
@Html.ActionLink("Link Text",
"AnotherAction")
```

In the preceding syntax, `AnotherAction` is the name of the action method that acts as the target for the hyperlink, and `Link Text` is the text to be displayed as the link.

Assuming that the preceding hyperlink is contained in a view associated with the `Home` controller, the preceding syntax will produce the following markup:

```
<a href="/Home/
AnotherAction">LinkText</a>
```

However, if you want to navigate to an action method in a different controller, you can consider the following syntax:

```
@Html.ActionLink("Link Text",
"AnotherAction", "AnotherController")
```

In the preceding syntax, `AnotherAction` is the name of the action method that acts as the target for the hyperlink, `Link Text` is the text to be displayed as the link, and `AnotherController` is the name of the controller that contains the action method, `"AnotherAction"`.

Html.BeginForm() and Html.EndForm()

The `Html.BeginForm()` method marks the start of a form. This helper method coordinates with the routing engine to generate a proper URL. The `Html.BeginForm()` helper method outputs the opening `<form>` tag. The `Html.BeginForm()` helper method has the following syntax:

```
@{Html.BeginForm
("MyAction","MyController");}
```

The preceding syntax will produce the following markup:

```
<form action="Mycontroller/MyAction"
method="post">
```

In the preceding syntax, `MyController` is the name of the controller and `MyAction` is the name of the action method to which the form will be posted.

Once created, you need to close the `<form>` tag by rendering a closing `</form>` tag. The `Html.EndForm()` helper method renders the closing `</form>` tag. The `Html.EndForm()` helper method provides a way to end the form created by using the `Html.BeginForm()` helper method. The `Html.EndForm()` helper method has the following syntax:

```
@{Html.EndForm();}
```

However, to avoid using the `Html.EndForm()` helper method to close the form, you can make use of the following syntax:

```
@using (Html.BeginForm
("MyAction","MyController"))
{
//markup to generate form elements
}
```

In the preceding syntax, an `@using` statement has been used with the `Html.BeginForm()` method. The preceding syntax will produce the following markup:

```
<form action="Mycontroller/MyAction">
<!--markup to generate the form
elements-->
</form>
```

The advantage of using `@using` before the `Html.BeginForm()` method is that it automatically renders a closing `</form>` tag and you do not need to explicitly include the `Html.EndForm()` method to add the closing `</form>` tag.

Html.Label() and Html.LabelFor()

The `Html.Label()` helper method is used to render an HTML `<label>` element, which refers to the specified model property. The `Html.Label()` helper method has the following syntax:

```
@Html.Label("propertyname")
```

For example, with reference to the `Product` class, the name of the product can be displayed by using the following statement.

```
@Html.Label("name")
```

The preceding statement will generate the following markup:

```
<label for="name">name</label>
```

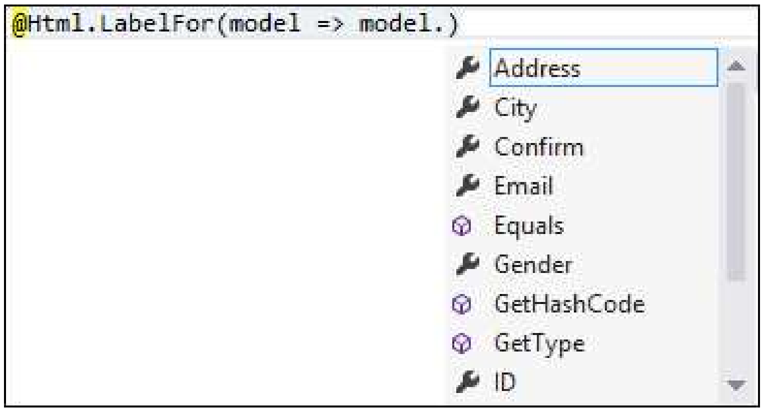
However, while writing the property name in the `Label()` method, there are chances that you may misspell the name of the property. Such mistakes would not be detected at the time of compilation of the code. To overcome such problems, you can use the `Html.LabelFor()` helper method.

The `Html.LabelFor()` helper method is the strongly typed version of the Html.Label() helper method. The `Html.LabelFor()` helper method has the following syntax:

```
@Html.LabelFor(model =>
model.propertyname)
```

The `Html.LabelFor()` helper method uses a lambda expression as its parameter, which provides compile time checking and ensures that the correct property name is specified.

In addition, it gives you the benefit of intellisense while trying to access the model properties, as shown in the following figure.



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

The Html.LabelFor() Helper Method

This helps in avoiding the chance of misspelling the name of the property.

Similar to the `Html.Label()` helper method, the `Html.LabelFor()` helper method renders an HTML `<label>` element with the name of the property.

For example, with reference to the `Product` class, the name of the product, the `name` property, can be displayed by using the following statement:

```
@Html.LabelFor(model => model.name)
```

Html.DisplayNameFor() and Html.DisplayFor()

The `Html.DisplayNameFor()` helper method is used to display the names of model properties. The `Html.DisplayNameFor()` helper method has the following syntax:

```
@Html.DisplayNameFor(model =>
model.propertyname)
```

In the preceding syntax, `propertyname` is the name of the property defined in the model class. For example, with reference to the `Product` class, the name of the product can be displayed by using the following statement:

```
@Html.DisplayNameFor(model =>
model.name)
```

The preceding statement will display the following output:

name

The `Html.DisplayFor()` helper method is used to display the values of the model properties. The `Html.DisplayFor()` helper method has the following syntax:

```
@Html.DisplayFor(model                    =>
model.propertyname)
```

In the preceding syntax, `propertyname` is the name of the property defined in the model class. Assuming that the preceding syntax is used for displaying the name of a product defined in the model class, the preceding syntax would be written as the following statement:

```
@Html.DisplayFor(model => model.name)
```

Html.TextBox() and Html.TextBoxFor ()

The `Html.TextBox()` helper method renders an input tag with the type attribute set to text. The `Html.TextBox()` helper method is used to accept free-form input from a user. The `Html.TextBox()` helper method has the following syntax:

```
@Html.TextBox("propertyname",
Model.propertyname)
```

In the preceding syntax, `propertyname` is the name of the property defined in the model class. The preceding syntax will generate the following markup:

```
<input                  id="propertyname"
name="propertyname"             type="text"
value="val" />
```

In the preceding markup, `val` is the value associated with the model property, `propertyname`.

The `Html.TextBoxFor()` helper method is the strongly typed version of the `Html.TextBox()` helper method. The `Html.TextBoxFor()` helper method has the following syntax:

```
@Html.TextBoxFor(model                    =>
model.propertyname)
```

The preceding syntax generates the same markup as that of the `Html.TextBox()` helper method.

Html.TextArea() and Html.TextAreaFor()

The `Html.TextArea()` helper method is used to render a `<textarea>` element for multi-line text entry. The `Html.TextArea()` helper method enables you to specify the number of columns and rows to display in order to control the size of the text area. The `Html.TextArea()` helper method has the following syntax:

```
@Html.TextArea("propertyname",
Model.propertyname, row, column, null)
```

In the preceding syntax, `row` and `column` specify the number of rows and columns. `propertyname` specifies the name of the model property. The fifth parameter in the preceding syntax specifies additional HTML attributes for the text area. In this case, it has been specified as `null`.

To include a text area with 5 rows and 20 columns for the `description` property of the `Product` model, you can write the following statement:

```
@Html.TextArea("description",
Model.description, 5, 20, null)
```

Assuming that the `description` property of the `Product` model includes the value, `Newly launched with the touch screen`, the preceding statement will generate the following markup:

```
<textarea   cols="20"   id="description"
name="description"    rows="5">    Newly
launched with touch screen</textarea>
```

The`Html.TextAreaFor()` helper method is the strongly typed version of the `Html.TextArea()` helper method. It has the following syntax:

```
@Html.TextAreaFor(model                    =>
model.propertyname,    rows,    columns,
null)
```

The `Html.TextAreaFor()` helper method generates the same markup as that of the `Html.TextArea()` helper method.

Html.EditorFor()

The `Html.EditorFor()` helper method is used to display an editor for the specified model property. The `Html.EditorFor()` helper method has the following syntax:

```
@Html.EditorFor(model                    =>
model.propertyname)
```

In the preceding syntax, `propertyname` is the name of the property defined in the model class. For example, to edit the value associated with a `name` field, the preceding syntax can be written as the following statement.

```
@Html.EditorFor(model => model.name)
```

Assuming that the name field of the model contains the value, `Samsung 5676 Mobile`, the preceding statement will generate the following markup:

```
<input   class="text-box   single-line"
id="address" name="address" type="text"
value="Samsung 5676 Mobile" />
```

Html.Password() and Html.PasswordFor()

The `Html.Password()` helper method renders a password field. The `Html.Password()` helper method has the following syntax:

```
@Html.Password
("UserPassword",Model.UserPassword)
```

In the preceding syntax, `UserPassword` is the name of the model property.

The preceding syntax will generate the following markup:

```
<input id="UserPassword"
name="UserPassword" type="password" />
```

The `Html.PasswordFor()` helper method is the strongly typed version of the `Html.Password()` helper method. It is used to render a password field. The `Html.PasswordFor()` helper method has the following syntax:

```
@Html.PasswordFor(model =>
model.UserPassword)
```

Html.Hidden() and Html.HiddenFor()

The `Html.Hidden()` helper method renders a hidden input. The `Html.Hidden()` helper has the following syntax:

```
@Html.Hidden("propertyname",
Model.propertyname)
```

In the preceding code snippet, `propertyname` is the name of the property.

The preceding syntax will generate the following markup:

```
<input id="propertyname"
name="propertyname" type="hidden"
value="val" />
```

In the preceding markup, val is the value of the model property, `propertyname`.

The `Html.HiddenFor()` helper method is the strongly typed version of the `Html.Hidden()` helper method. The `Html.HiddenFor()` helper method has the following syntax:

```
@Html.HiddenFor(model =>
model.propertyname)
```

In the preceding syntax, `propertyname` is the name of the model property that needs to be added to a hidden field.

Html.CheckBox() and Html.CheckBoxFor()

The`Html.CheckBox()` helper method renders a check box input element that enables the user to select a true or false condition. The`Html.CheckBox()` helper method has the following syntax:

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

The preceding syntax will generate the following markup:

```
<input id="propertyname"
name="propertyname" type="checkbox"
value="true" /> <input
name="propertyname" type="hidden"
value="false" />
```

In the preceding markup, the checkbox helper method renders a hidden input in addition to the checkbox input. This is required because a browser submits a value for a checkbox only when the checkbox is selected. The hidden input guarantees that a value will be submitted, even if the user does not select the checkbox.

The `Html.CheckBoxFor()` helper method is the strongly typed version of the `Html.CheckBox()` helper method. The `Html.CheckBoxFor()` helper method has the following syntax:

```
Html.CheckBoxFor(model =>
model.propertyname)
```

Html.DropDownList() and Html.DropDownListFor()

The `Html.DropDownList()` helper method returns 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. The `Html.DropDownList()` helper method has the following syntax:

```
@Html.DropDownList("myList", new
SelectList(new [] {"A", "B", "C"}),
"Choose")
```

In the preceding syntax, `A`, `B`, and `C` are the options available in the drop down list. `Choose` is the default value at the top of the list, indicating to the user that a value needs to be chosen from the drop-down list.

Consider an example where you want to create a drop-down list named City with the values Paris, Tokyo, and Rome populated in it. For this, you can use the following statement:

```
@Html.DropDownList("City", new
SelectList(new [] {"Paris", "Tokyo",
"Rome"}), "Choose")
```

The preceding statement will generate the following markup:

```
<select id="City" name="City">
<option value="">Choose</option>
<option>Paris</option>
<option>Tokyo</option<
<option>Rome</option>
</select>
```

The `Html.DropDownListFor()` helper method is the strongly typed version of the `Html.DropDownList()` helper method. The `Html.DropDownListFor()` helper method has the following syntax:

```
@Html.DropDownListFor(model              =>
model.propertyname,  new  SelectList(new
[] {"A", "B","C"}),"Choose")
```

Html.RadioButton() and Html.RadioButtonFor()

The `Html.RadioButton()` helper method is used 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. The `Html.RadioButton()` helper method has the following syntax:

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

In the preceding syntax, `name` is the name of the radio button input element, `value` is the value associated with a particular radio button option, and `isChecked` is a Boolean value that indicates whether the radio button option is selected or not.

For example, to accept the gender of a user, you can create a collection of two radio buttons, as shown in the following code snippet.

```
Male         @Html.RadioButton("Gender",
"Male", true)
Female       @Html.RadioButton("Gender",
"Female")
```

The preceding code snippet will generate the following markup:

```
Male <input checked="checked"
id="Gender" name="Gender" type="radio"
value="Male" />
Female <input id="Gender"
name="Gender" type="radio"
value="Female" />
```

The `Html.RadioButtonFor()` helper method is the strongly typed version of the `Html.RadioButton()` helper method. The `Html.RadioButtonFor()` helper 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. The `Html.RadioButtonFor()` helper method has the following syntax:

```
@Html.RadioButtonFor(model =>
model.propertyname, "Value")
```

In the preceding syntax, `propertyname` is the property of the model class that needs to be rendered.

Url.Action()

The `Url.Action()` helper method will return the URL for the specified action within a controller. The following syntax can be used to build a URL from a given controller and action:

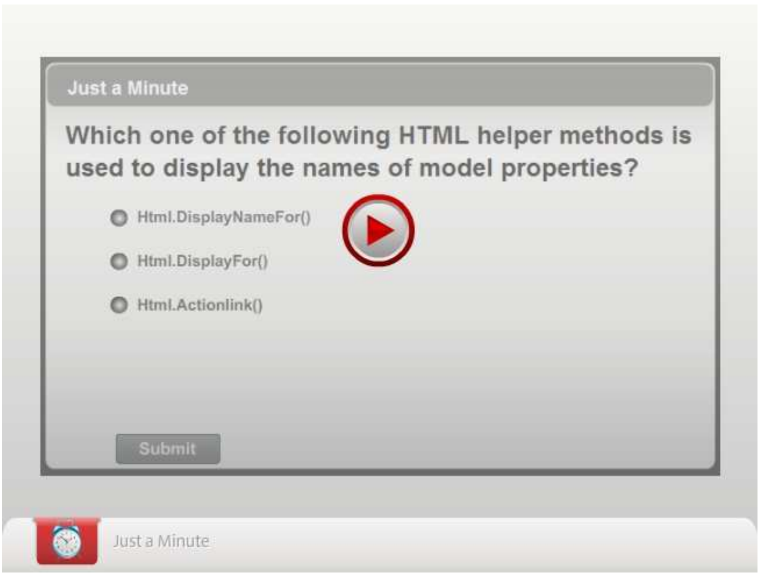
```
@Url.Action("ActionName",
"ControllerName")
```

For example, to generate a URL of the Index action in the Home controller, the following statement can be used:

```
@Url.Action("Index", "Home")
```

The preceding statement will generate the following output:

/Home/Index


Just a Minute

Which one of the following HTML helper methods is used to display the names of model properties?

- Html.DisplayNameFor()
- Html.DisplayFor()
- Html.Actionlink()

Summary

In this chapter, you learned that:

- ❑ In an MVC application, the component that enables interaction with data is called model.
- ❑ A model consists of a set of classes, where the objects of the classes represent the data associated with the application.
- ❑ The model data passed could be a single object or a collection of objects.
- ❑ Scaffolding provides the following templates for creating controller and associated views:
  - Empty controller
  - Controller with empty read/write actions
  - API controller with empty read/write actions
  - Controller with read/write actions and views using Entity Framework
- ❑ Scaffolding provides the following templates for creating views:
  - List
  - Create
  - Edit
  - Details
  - Delete
- ❑ Some of the helper methods that can be used in building an MVC application are:

- `Html.ActionLink()`
- `Html.BeginForm()` and`Html.EndForm()`
- `Html.Label()` and`Html.LabelFor()`
- `Html.DisplayNameFor()` and`Html.DisplayFor()`
- `Html.TextBox()` and`Html.TextBoxFor()`
- `Html.TextArea()` and`Html.TextAreaFor()`
- `Html.EditorFor()`
- `Html.Password()` and`Html.PasswordFor()`
- `Html.HiddenFor()`
- `Html.CheckBox()` and`Html.CheckBoxFor()`
- `Html.DropDownList()` and`Html.DropDownListFor()`
- `Html.RadioButton()` and`Html.RadioButtonFor()`
- `Url.Action()`

Chapter 4

Validating Data

In an MVC application, a model represents classes that interact with data. These classes contain various properties, which define the structure of the data. To preserve this structure, you need to maintain the integrity of the model. However, while interacting with a view, it is possible that users enter invalid data. Therefore, you need to validate the data entered by users.

Validation is the process of checking the input data against certain criteria. In an MVC application, the data entered by a user in a form is checked against the structure of the model class.

In order to validate the data, there are various data annotations provided by MVC, which help you to implement validation.

This chapter discusses how to work with data annotations and implement data validation.

Objectives

In this chapter, you will learn to:

- ❑ Identify Data Annotations
- ❑ Implement Validation

Introduction to Data Annotations

Consider the scenario of the Newbay shopping store. This store has a website that allows customers to search the products and place orders online. However, before performing any such activity, customers need to create an account on the website. For this purpose, they need to fill the registration form. While filling this form, the customer needs to enter certain details, such as name, gender, age, and address.

The customer details, such as name, gender, 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 customer submits the form, the details of the customer must be validated. If the validation criteria are valid, the form is accepted. Otherwise, an error message is displayed to the customer. Such type of validation can be implemented on a website by using various data annotations available in the MVC framework.

The advantage of using data annotations is that they enable you to perform validation efficiently by adding one or more data annotation attributes to a model property. As a result, validation is easily performed.

These data annotations are available in the `System.ComponentModel.DataAnnotations` namespace. Therefore, before using any data annotation, you need to add this namespace in your application. Some of the commonly used data annotations are:

- ❑ `Required`
- ❑ `StringLength`
- ❑ `RegularExpression`
- ❑ `Range`
- ❑ `Compare`
- ❑ `Display`
- ❑ `ReadOnly`
- ❑ `DataType`
- ❑ `ScaffoldColumn`

Let us discuss these annotations individually.

Required

The `Required` data annotation attribute specifies that the property, with which this annotation is associated, is a required property. 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.

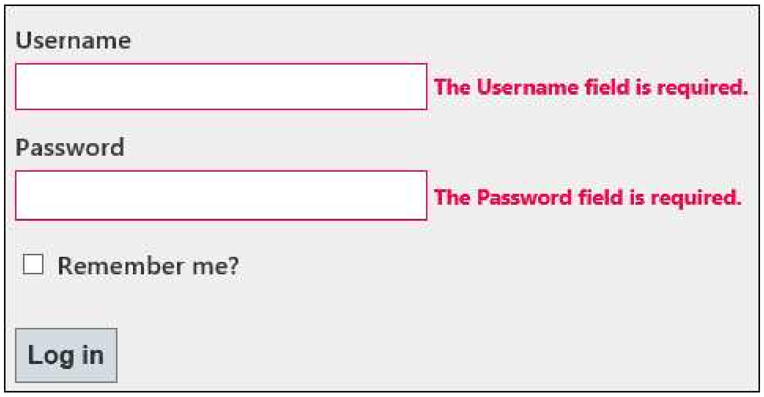
Consider a scenario where you need to ensure that a user enters values for the username and the password fields, before submitting the login Web page. To accomplish this task, you can use the `Required` attribute while declaring the `Username` and `Password` properties in the Model class, as shown in the following code snippet:

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

In the preceding code snippet, the `[Required]` attribute is used before a property to validate the `Username` and `Password` properties of a class. This attribute raises a validation error if the associated property value is either null or empty. The validation errors raised through the data annotation attributes can be displayed to the user by using validation helpers.

NOTE *You will learn about validation helpers in the next section.*

By using data annotation and validation helpers, a default message is displayed to the user, as shown in the following figure:


Username

The Username field is required.

Password

The Password field is required.

The Output Showing Usage of the Required Attribute

In the preceding figure, the `[Required]` attribute raises a default error message. However, you can also customize error messages. The following code snippet illustrates how to add a custom error message to the `[Required]` attribute:

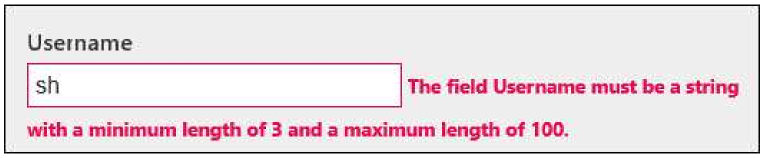
```
[Required(ErrorMessage = "Please enter
the Username")]
public string Username { get; set; }
[Required(ErrorMessage = "Please enter
the Password")]
public string Password { get; set; }
```

StringLength

The `StringLength` data annotation attribute is used to specify the minimum and maximum lengths of a string field. The attribute raises a validation error, if the user enters a string with more or less number of characters than the specified range. For example, consider the following code snippet:

```
[StringLength(100, MinimumLength = 3)]
public string Username {get; set;}
```

In the preceding code snippet, the maximum length of the `Username` field is set to `100`, and the minimum length is set to `3`. The `MinimumLength` parameter is optional for the `StringLength` attribute. If a user does not enter the string in the specified range, a default error message is displayed, as shown in the following figure:


Username

sh

The field Username must be a string with a minimum length of 3 and a maximum length of 100.

The Username Field

However, if you want to display a custom message, you can use the following code snippet:

```
[StringLength(100,ErrorMessage = "Name
cannot be more than 100 characters
long")]
public string Username {get; set;}
```

Each annotation attribute allows you to pass an optional parameter named ErrorMessage that can be used to specify a custom error message associated with the attribute.

RegularExpression

At times, you need to accept user input in a specific text pattern. For example, if you want to accept an email address, it is mandatory to have the @ symbol between the user name and the domain name. For this, you can use a regular expression annotation attribute. It allows you to match a text string with a search pattern. The pattern consists of one or more character literals, operators, or constructs. The regular expression has the following syntax:

```
[RegularExpression("pattern")]
```

To check the validity of an email address, you can use the following regular expression pattern:

```
[RegularExpression(@"[A-Za-z0-9._%+-]
+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]
public string Email { get; set; }
```

In the preceding code snippet, the regular expression, `[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}`, is used to describe an email address.

An email address is divided into three segments arranged in the following manner:

Segment1@Segmentt2.Segment3

Similarly, there are three segments in the given regular expression:

Segment1: `[A-Za-z0-9._%+-]+`

Segment2: `[A-Za-z0-9.-]+`

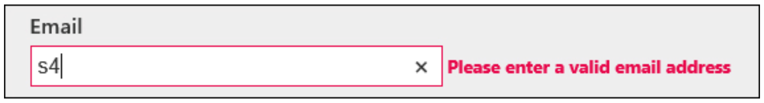
Segment3: `[A-Za-z]{2,4}`

The characters and character ranges specified within the square brackets in each segment specify the types of characters that can appear in that segment.

The + sign at the end of Segment1 and Segment2 indicates that these segments can consist of one or more characters of the types included within the square brackets preceding the + sign.

`{2, 4}` at the end of Segment3 indicates that the third segment can include 2-4 characters.

If a user does not enter the email in the correct format, the following error message is displayed.


Email

s4

Please enter a valid email address

The Email field

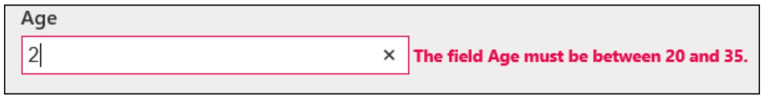
Similarly, you can also create regular expressions to match other text patterns.

Range

You can use the Range attribute to specify the minimum and maximum constraints for a numeric value. For example, if you want the user to enter the age between `20` and `35`, you can use the Range attribute, as shown in the following example:

```
[Range (20, 35)]
public int Age { get; set; }
```

In the preceding example, the first parameter, `20`, specifies the minimum value, and the second parameter, `35`, specifies the maximum value. Here, the minimum and maximum values are inclusive. The Range attribute will display the following error if the age entered is not in the specified range, as shown in the following figure.



Age

2

The field Age must be between 20 and 35.

The Age field

You can specify the range for double values too, as shown in the following example:

```
[Range(typeof(decimal), "0.00",
"49.99")]
public decimal Price { get; set; }
```

In the preceding example, the first parameter, `0.00`, specifies the minimum value, and the second parameter, `49.99`, specifies the maximum value for the Price property.

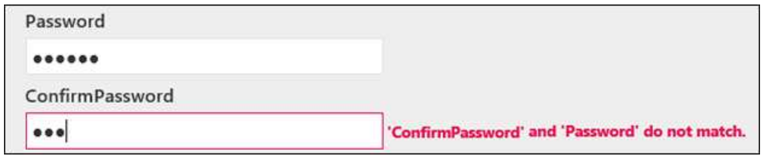
Compare

At times, you have two fields in a form, in which you want to accept the same value. For example, you want that a user should enter the same password in both, the `Password` and `ConfirmPassword` fields. Therefore, to match the value of the two fields in a form, you can use the `Compare` attribute. This ensures that the two properties on a model object have the same value. Consider the following example:

```
public string Password { get; set; }
 [Compare("Password")]
public string ConfirmPassword { get;
set; }
```

In the preceding example, the `ConfirmPassword` field is checked to ensure that it contains the same value as that of the `Password` field.

If a user does not enter the same password in both the fields, an error message is displayed, as shown in the following figure.



Password

••••••

ConfirmPassword

•••

'ConfirmPassword' and 'Password' do not match.

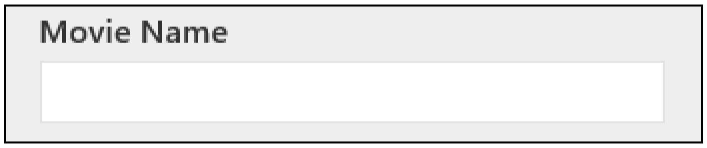
The Password and the ConfirmPassword fields

Display

When you use scaffolding to create views, the name of each property is displayed as a label for the fields corresponding to that property. The `Display` attribute allows you to specify a user friendly display name for a model property. For example, consider the following code snippet:

```
[Display(Name = "Movie Name")]
public string Movie_Name { get; set; }
```

In the preceding code snippet, the name of the property is `Movie_Name`. However, the name that will be displayed on the view is `Movie Name`, as shown in the following figure.



Movie Name

The Movie_Name field

ReadOnly

At times, you want to display a read-only field on a form to the user. Consider the example of a form that allows a user to enter a product code, price, and quantity of items. Once the user specifies the details, the total amount is calculated and displayed to the user in the `TotalAmount` field. Because the value of the `TotalAmount` field is a calculated value, the users need not calculate and enter the value of the total amount on their own. Therefore, you can specify the `TotalAmount` field as read-only by using the `ReadOnly` attribute. This attribute ensures that the default model binder will not set the property with a new value from the request. Consider the following example:

```
[ReadOnly(true)]
public Int32 TotalMarks { get; set; }
```

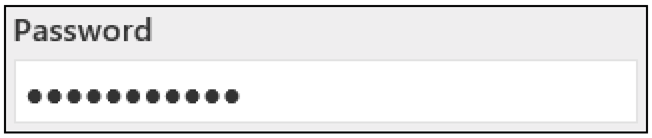
In the preceding example, the `TotalMarks` property is set to read-only. If you use an `EditorFor()` helper for the `TotalMarks` property, an enabled input will be displayed and the user will be able to enter a value in the input element. However, the model binder will not set the property with a new value from the request.

DataType

The DataType attribute helps you to provide information about the specific purpose of a property at run time. For example, a property of the string type can be used to hold an email address, a URL, or a password. By applying DataType for a Password field as Password, the HTML helpers in ASP.NET MVC will render an input element with a type attribute set to Password. Therefore, upon entering a value in the Password field, the user sees a special character, such as a dot or star, instead of the original character. Consider the following example:

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

In the preceding example, DataType of the Password field is set to Password, which displays a dot, instead of the original character, as shown in the following figure.

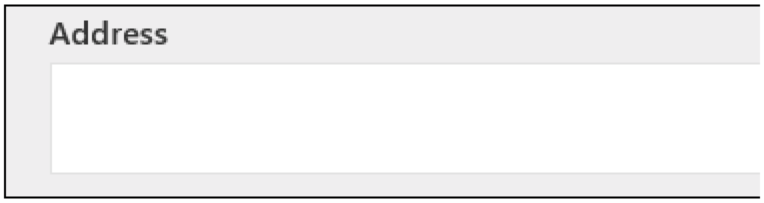


The Password Field

You can also add other DataType attributes such as Currency, Date, Time, and MultilineText. To add MultilineText to an Address field, consider the following example:

```
[DataType(DataType.MultilineText)]
public string Address { get; set; }
```

In the preceding example, DataType of the Address field is set to MultilineText, which displays a text area, as shown in the following figure.



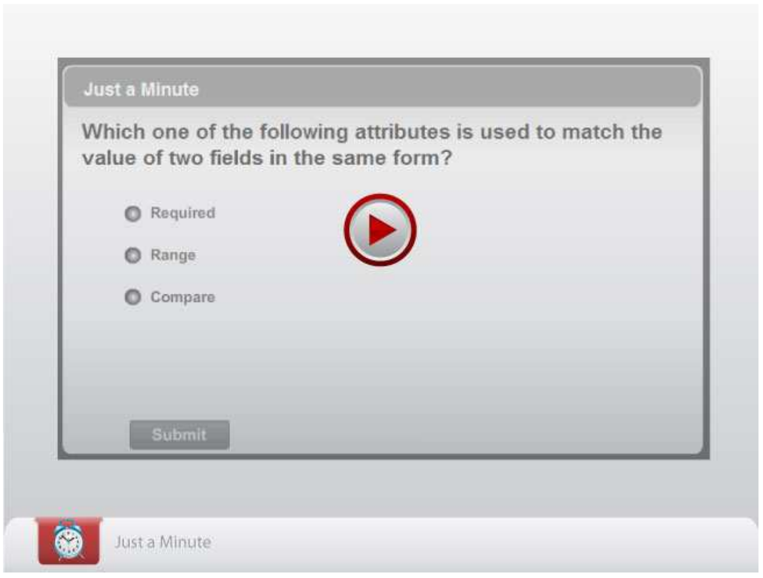
The Address Field

ScaffoldColumn

At times, you do not want to render a particular field on a view. For example, consider that you have the Age property in your class. You want that when a user fills a form in a browser, the label and input for the Age field should not be displayed on the browser. Consider the following example:

```
[Required]
[ScaffoldColumn (false)]
public string Age { get; set; }
```

With the preceding attribute in place, scaffolding will not add the the Age column to any view based on the model.



Just a Minute

Which one of the following attributes is used to match the value of two fields in the same form?

- Required
- Range
- Compare

Implementing Validation

Consider the scenario of the Newbay Web application. A user needs to enter values for the username and password fields in a Login form before accessing the website. Once the user enters the username and the password and submits the login form, it is sent to the server that validates the entered username and the password. In case the username or the password contains a blank value, a value in an incorrect format, or a value that does not match the credentials stored in the database, the server sends an error message to the client.

To check whether the credentials provided by the user match the credentials stored in the database, a round-trip to the server is required. However, the other details, such as checking for blank values and correctly formatted values, can even be done at the client side. Therefore, to avoid round trips to a server, you can implement validation at the client- side.

Client-side validation provides immediate feedback without submitting anything to the server. It is usually implemented using JavaScript. The data that the user enters is validated before being sent to the server. This provides immediate feedback and an opportunity to correct any error before the data is sent to the server. However, if JavaScript is disabled at the client-side, then the validation is only done at the server-side.

Both the client-and-server side validations, display a default message for an incorrect value. However, you can change this default message by specifying the message in the corresponding annotation attribute. In addition, you can summarize the entire validation message at one place, on a Web page. The error messages can be displayed by using HTML validation helpers.

Let us see how to use these helpers to display error messages.

Using HTML Helpers to Display Error Messages

In the registration form of the Newbay application, a user needs to register personal details, such as user name, password, email, city, and address. While filling the form, if the user enters an incorrect value in any of the fields, an error message needs to be displayed to the user. To accomplish this task, MVC provides the following two main helper methods:

- `Html.ValidationMessageFor()`
- `Html.ValidationSummary()`

Html.ValidationMessageFor

Consider the scenario where a registration form needs to be filled. While filling this form, if a user enters an incorrect value in any of the fields, an error message is displayed separately for all the fields. To display these error messages, the`ValidationMessageFor()` HTML helper is used with each field. For example, consider the following model class:

```
public class Customer
 {
 public long ID { get; set; }
 [Required]
 public string Name { get; set; }
 [Required]
 [DataType(DataType.Password)]
 public string Password { get; set; }
 [Required]
 [Compare("Password")]
 [DataType(DataType.Password)]
 public string Confirm { get; set; }
 [Required]
 [RegularExpression(".+\\@.+\\..+")]
 public string Email { get; set; }
 [Required]
 public string City { get; set; }
 [Required]
 [DataType(DataType.MultilineText)]
 public string Address { get; set; }
```

In the preceding code snippet, the model properties, such as `Name`, `Password`, `Confirm`, `Email`, `City`, and `Address`, have different data annotations applied to them.

Based on the properties of the preceding model class, scaffolding will generate the following code for the registration form:

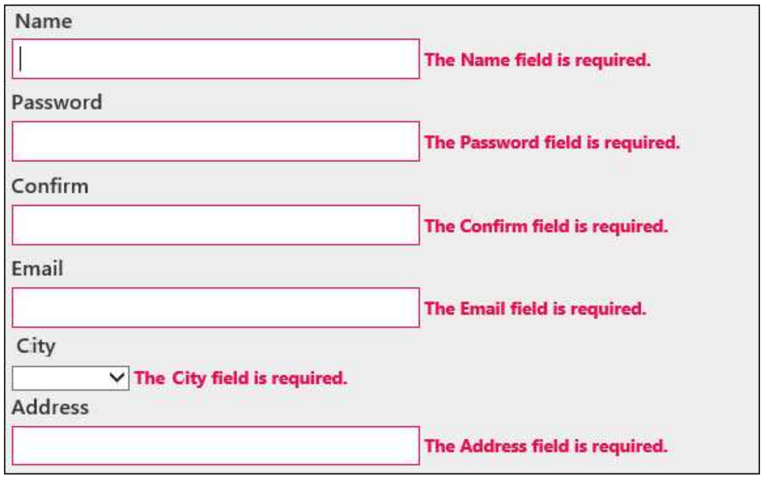
```
@using (Html.BeginForm()) {
 @Html.ValidationSummary(true)
 <fieldset>
  <legend>Customer</legend>
 <div class="editor-label" >
  @Html.LabelFor(model => model.Name)
 </div>
<div class="editor-field" >
@Html.EditorFor(model => model.Name)
@Html.ValidationMessageFor(model =>
model.Name)
</div>
<div class="editor-label" >
@Html.LabelFor(model =>
model.Password)
</div>
<div class="editor-field" >
@Html.EditorFor(model =>
model.Password)
@Html.ValidationMessageFor(model =>
model.Password)
</div>
<div class="editor-label" >
@Html.LabelFor(model => model.Confirm)
</div>
<div class="editor-field" >
@Html.EditorFor(model =>
model.Confirm)
@Html.ValidationMessageFor(model =>
model.Confirm)
</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>
<div class="editor-label" >
@Html.LabelFor(model => model.City)
</div>
<div class="editor-field" >
@Html.DropDownListFor
(model=>model.City,(SelectList)
ViewBag.category,string.Empty)
@Html.ValidationMessageFor(model =>
model.City)
</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>
</fieldset>
@section Scripts {
@Scripts.Render("~/bundles/jqueryval")
}
```

In the preceding code snippet, the

`@Html.ValidationMessageFor()` helper method is used with each form field to display the default error messages related to that field.

In the preceding code snippet, the `@section` directive is used to include the javascript libraries required for implementing validation. You will learn more about the `@section` directive in the subsequent chapters.

If a user submits the registration form without filling the values in the fields, the preceding code snippet will render the following interface:

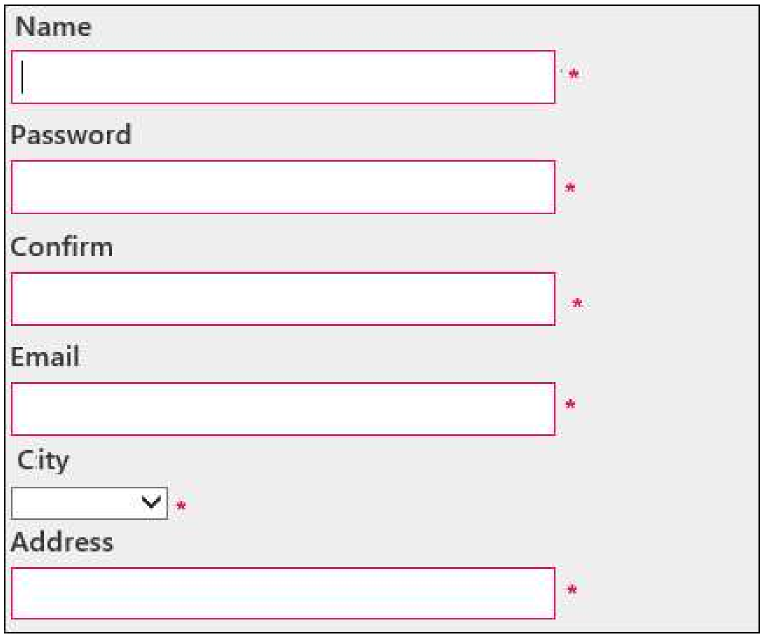

Name
The Name field is required.
Password
The Password field is required.
Confirm
The Confirm field is required.
Email
The Email field is required.
City
The City field is required.
Address
The Address field is required.

The Output Showing the Default Error Along with the Fields

However, you can replace the default messages for the individual fields by short messages or a special character, such as asterisk (*). In order to implement this functionality, you need to pass the short message as the second parameter of the `Html.ValidationMessageFor()` helper method, as shown by the highlighted part of the following code snippet:

```
@using (Html.BeginForm()) {
 @Html.ValidationSummary(true)
 <fieldset>
  <legend>Customer</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.Password)
 </div>
 <div class="editor-field">
 @Html.EditorFor(model =>
 model.Password)
 @Html.ValidationMessageFor(model =>
 model.Password, "*")
</div>
<div class="editor-label">
 @Html.LabelFor(model => model.Confirm)
</div>
<div class="editor-field">
 @Html.EditorFor(model =>
 model.Confirm)
 @Html.ValidationMessageFor(model =>
 model.Confirm, "*")
</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>
<div class="editor-label">
 @Html.LabelFor(model => model.City)
</div>
<div class="editor-field">
 @Html.DropDownListFor
 (model=>model.City,(SelectList)
ViewBag.category,string.Empty)
 @Html.ValidationMessageFor(model =>
 model.City, "*")
</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>
</fieldset>
}
```

If a user submits the registration form without filling the values in the fields, the preceding code snippet will render the following output:



The Output Showing an Asterisk Character Instead of the Default Error Message

> NOTE
> *A short error message is usually displayed with the form fields when you want to display the detailed error messages together at one place by using the `Html.ValidationSummary()` helper method.*

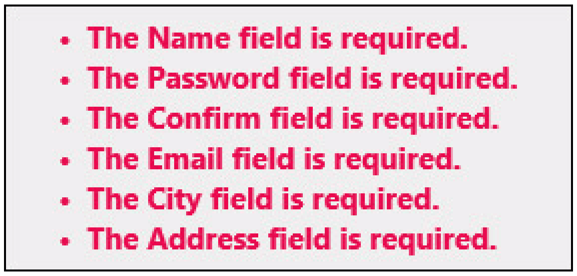
Html.ValidationSummary

The`Html.ValidationFor()` helper can be associated with a single form field, and it will display the errors associated with that field. Therefore, you need a separate `ValidationFor()` helper for each form field. If you want to summarize all the error messages at one place, you can use the `Html.ValidationSummary()` helper. In order to implement this functionality, you need to add the highlighted code snippet in the view file, as shown in the following code:

```
@using (Html.BeginForm()) {
 @Html.ValidationSummary(false)
```

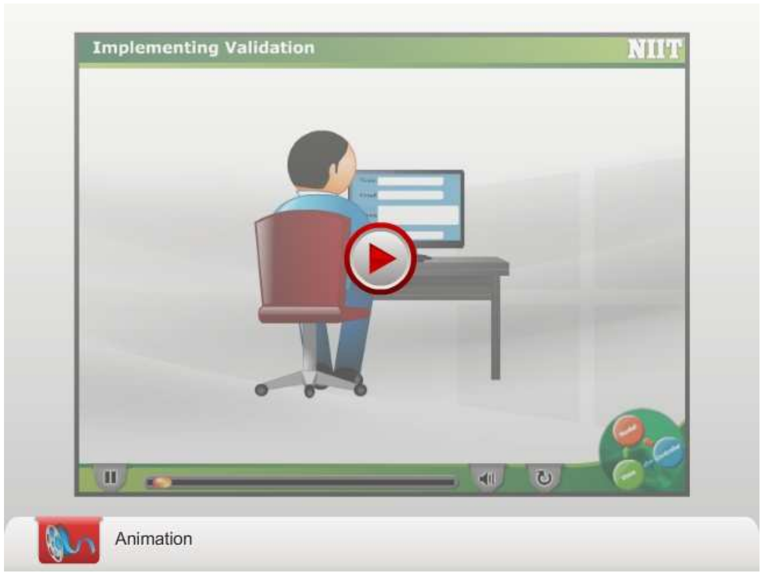
In the preceding code snippet, the value, `false`, indicates that the validation messages for each individual property should not be excluded while displaying the validation summary. As a result, the validation messages for all the properties will be displayed together in the validation summary.

The validation summary for the preceding example will show the summarized error messages, as shown in the following figure.



- The Name field is required.
- The Password field is required.
- The Confirm field is required.
- The Email field is required.
- The City field is required.
- The Address field is required.

The Summarized Error Messages



Introduction to ModelState

Consider a situation where a user needs to enter the username and password before accessing a Web page. Once the user enters the password in the correct format, the entered password will be bound with the password property in the model class. This process is known as model binding. However, if the user enters the password in an incorrect format, the process of binding the password with the password property of the model class fails. This results in an error. The error message associated with this error can be displayed by using the `ValidationMessage` helper method. If the model binding succeeds, the `ModelState.IsValid` property returns true. However, if there is any error in the model state, the`ModelState.IsValid` property returns false.

Consider the following example:

```
if (ModelState.IsValid)
{
//Code to save the entered data and
display an appropriate view
}
else
{
//Code to redisplay the view with
errors
}
```

In the preceding code snippet, if `ModelState.IsValid` is true, the data entered by the user is saved. Otherwise, the view is redisplayed with errors.

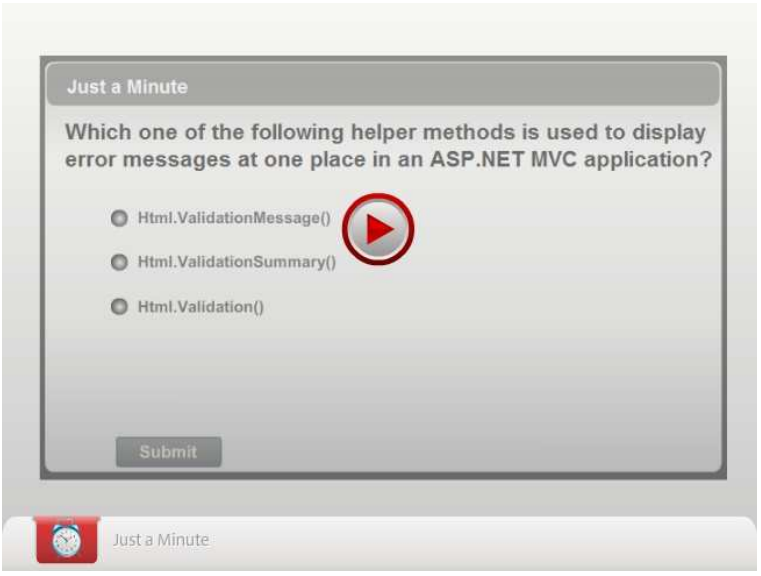
Adding Errors to the Model State Using Server-side scripting

The information entered by a user can be validated at client-side using various data annotations. However, there are times when this validation is not sufficient. For example, while creating an email account, a user needs to choose an email Id not taken before. For this, you need to validate the entered data at the server-side. MVC allows server-side validation of the entered data in the corresponding action method. Inside the action method, you can write the code to check for an error. If you find any error, using the `AddModelError()` method, you can add an error message in `ModelState`. The `AddModelError()` method adds a model error to the error collection of the model state dictionary.

Consider the following code snippet:

```
[HttpPost]
public ActionResult Create(Credentials
credentials)
{
 if (usernameAlreadyExists
 (credentials.Username))
  {
  ModelState.AddModelError("Username",
  "Username is not available.");
  }
 if (ModelState.IsValid)
  {
  /*Code to save the entered data and
  display an appropriate view */
  }
 else
  {
  //Code to redisplay the view with
  errors
  }
}
```

In the preceding code snippet, the `Username` property of the `credential` object is passed to a user-defined function, `usernameAlreadyExists()`, which checks whether the username provided by the user already exists or not. If the username already exists, an error message indicating the same is added to `ModelState`. If the model state contains an error, the `ModelState.IsValid` property returns false. However, if the model state does not contain any error, the `ModelState.IsValid` property returns a true value and the data entered by the user is saved.



Which one of the following helper methods is used to display error messages at one place in an ASP.NET MVC application?

- Html.ValidationMessage()
- Html.ValidationSummary()
- Html.Validation()

Activity 4.1: Implementing Validation

Summary

In this chapter, you learned that:

- Validation is the process of checking the input data against certain criteria.
- Some of the commonly used data annotations are:
  - `Required`
  - `StringLength`
  - `RegularExpression`
  - `Range`
  - `Compare`
  - `Display`
  - `ReadOnly`
  - `DataType`
- The `Required` data annotation attribute specifies that the property, with which this annotation is associated, is a required property.
- The `StringLength` data annotation attribute is used to specify the minimum and maximum lengths of a string field.
- You can use the `Range` attribute to specify the minimum and maximum constraints for a numeric value.
- The `Display` attribute allows you to specify a user friendly display name for a model property.
- The `DataType` attribute helps you to provide information about the specific purpose of a property at run time.
- Client-side validation provides immediate feedback without submitting anything to the server.

- Both the client-and-server side validations, display a default message for an incorrect value.
- MVC provides the following two main helper methods:
    - `Html.ValidationMessageFor()`
    - `Html.ValidationSummary()`
- The `Html.ValidationFor()` helper can be associated with a single form field, and it will display the errors associated with that field.

Chapter 5

Managing Data

Most Web applications handle data available as a database, text file, xml file, or spreadsheet. To access this data, an application needs to interact with the relevant data source. In an application, data is usually stored in the form of classes and objects. However, in a database, data is stored in the form of tables and views. Therefore, the definitions of classes or objects need to be mapped with the database tables or relational schemata. This requires you to write additional code. To overcome this problem, Entity Framework can be used. Entity Framework maps the classes and objects in an application to the tables in a relational data store. Moreover, it eliminates the need to write most of the data access code that needs to be written otherwise.

This chapter discusses the fundamentals of Entity Framework. In addition, it explains how to work with Entity Framework to manage data in an application.

Objectives

In this chapter, you will learn to:

- ❑ Identify the fundamentals of Entity Framework
- ❑ Work with Entity Framework

Introduction to Entity Framework

Consider the scenario of the Newbay store Web application. The application needs to display information about customers, such as customer ID and contact information. For this, the application uses the SQL database that contains the Customer table. In an application, data is usually stored in the form of classes and objects. However, in a database, data is stored in the form of tables and views. Therefore, in order to retrieve the details of customers, the definition of classes and objects needs to be mapped with that of the relational objects. This requires you to write complex code.

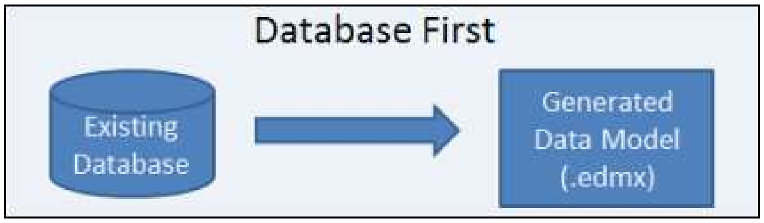
To overcome this problem, .NET provides the Object-Relational Mapping (ORM) framework called Entity Framework. Entity Framework enables you to work with relational data in the form of objects by representing the relational objects, such as tables, views, stored procedures, and functions, in the form of a conceptual model in the application. The conceptual model represents data as entities and relationships among these entities. In an Entity Framework-based application, the conceptual model is referred to as Entity Data Model (EDM). EDM allows you to work with data as entities or objects.

Entity Framework eliminates the need to write most of the data-access code that otherwise needs to be written. It uses different approaches to manage data related to an application. These approaches are:

- ❑ The database-first approach
- ❑ The model-first approach
- ❑ The code-first approach

The Database-first Approach

While developing an application, if you already have an existing database, the database-first approach is considered to be the best-suited approach. Entity Framework uses the database and generates the model for you, as depicted by the following figure.



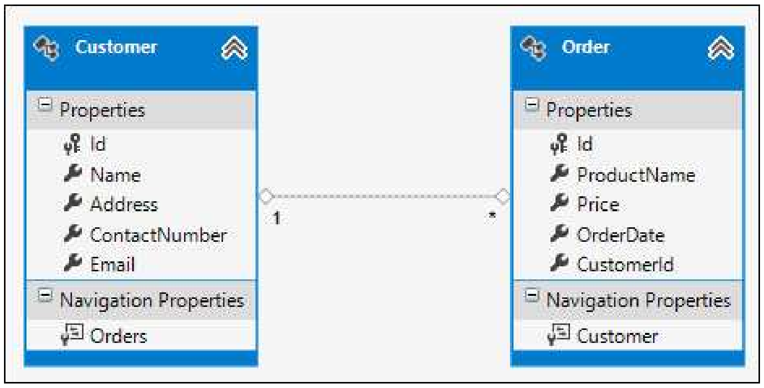
The Database-first Approach

The resulting data model will contain all the classes and properties corresponding to the existing database objects, such as tables and columns. The database structure and model related information are stored in an XML file with the **.edmx** extension. You can generate, edit, and update the **.edmx** file by using the Entity Framework designer provided by Visual Studio. The Entity Framework designer is a graphical designer tool that enables you to modify the **.edmx** file. This tool is used to visually create and change entities and associations.

The Model-first Approach

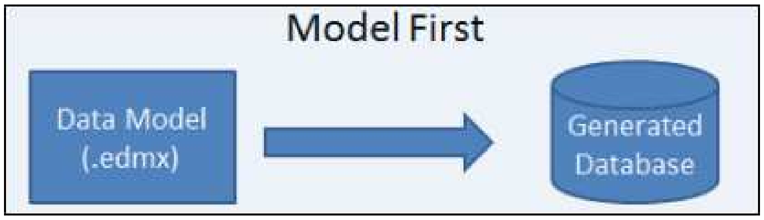
If you don't have a database and you are at the initial stages of development, you can follow the model-first approach. In the model-first approach, the first priority is the model, while the code and the database are secondary. The model-first approach gives you the flexibility to delay the creation of a database and, hence, allows you to view the different possibilities of improving its structure.

In the model-first approach, you can first create a model by using the Entity Framework designer. The resulting model and mapping information are stored in the **.edmx** file. The following figure displays a model created by using the Entity Framework designer.



A Model Created by Using the EF Designer

While creating a model, you can create entities, relationships between two entities, and inheritance hierarchies. Once the model is created, the Entity Framework designer uses the **.edmx** file of the model to create the database, as depicted by the following figure.



The Model-first Approach

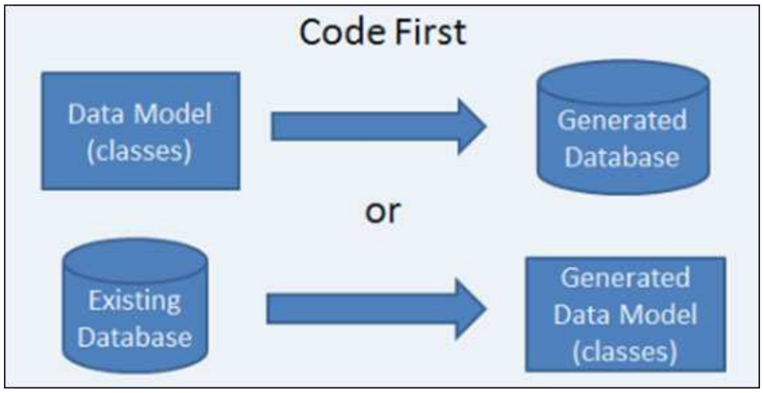
The Code-first Approach

In the code-first approach, you can start the development of your application by coding custom classes and properties, which correspond to tables and columns in the relational database. This approach allows you to build a database without using the Entity Framework designer or the **.edmx** file.

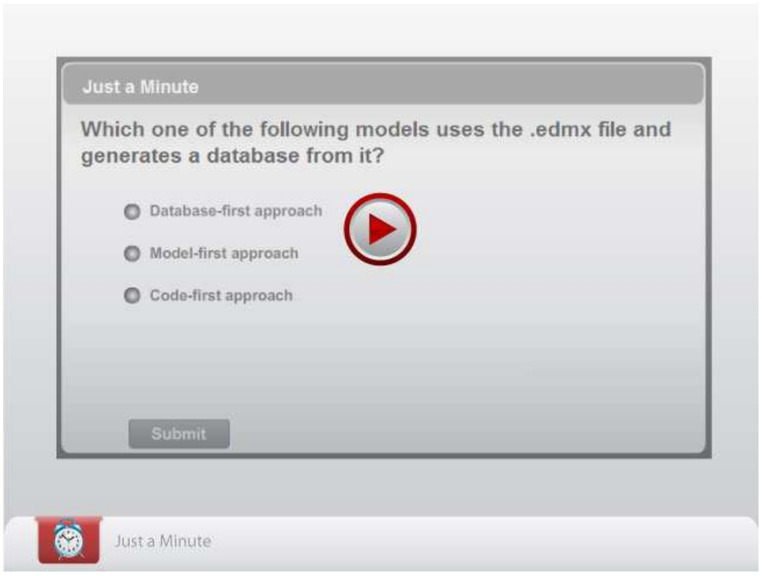
In the code-first approach, if you don't have an existing database, you first create your own custom classes, and then the database is created from them automatically. Further, if you change the model class, Entity Framework provides you the options to drop and recreate the database accordingly.

If you have an existing database, you can still define classes that map to the existing database. In addition, reverse engineering tools are available that allow you to generate the model classes from the existing database.

The following figure represents the code-first approach.



The Code-first Approach



Just a Minute

Which one of the following models uses the .edmx file and generates a database from it?

- Database-first approach
- Model-first approach
- Code-first approach

Working with Entity Framework

Entity Framework uses different approaches for development, such as code-first, database-first, and model-first, to manage data related to objects. This course focuses mainly on the code-first approach. The other two approaches are discussed in brief.

The code-first approach allows you to define your own domain model by creating custom classes. The database is then created by using this model. The structure of the database is determined by examining the structure of the model and using certain conventions.

Let us discuss the conventions used for the code-first approach. In addition, let us see how to implement the Entity Framework code-first approach.

Code First Conventions

The code-first approach allows you to provide the description of a model by using the C# or Visual Basic .NET classes. Based on these class definitions, the code-first conventions detect the basic structure of a model. The code-first conventions are a set of rules that automatically configure a conceptual model. These conventions are defined in the `System.Data.Entity.ModelConfiguration.Conventions namespace`. Some of the conventions are:

- **Table naming convention:** If you have created an object of the model class named Customer and want to store its data in the database, Entity Framework assumes that you want to store the data in the table named `Customers`.
- **Primary key convention:** If in a class, you have created the property named `ID` or `<classname>` ID, the property is assumed to be a primary key. Also, Entity Framework sets up an auto-incrementing key column in SQL Server to hold the property value.

  Consider the following code snippet:

  ```
  public class Customer
  {
  public int ID { get; set; }
  public string name {get; set;}
  }
  ```

  In the preceding code snippet, you have created the property, ID, inside the class named `Customer`. Therefore, it is inferred as a primary key.

  Consider another code snippet:

  ```
  public class Customer
  {
   public int CustomerID { get;
   set; }
   public string name {get; set;}
  }
  ```

  In the preceding code snippet, the `CustomerID` property will be inferred as a primary key. This is because the name of the property contains the class name followed by `ID`.
- **Relationship convention:** Entity Framework also follows certain conventions to identify a relationship between two entities. A relationship between two entities can be defined by using navigational properties. In addition to the navigational properties, it is recommended to define a foreign key property on types that represent dependent objects. For example, consider the following code snippet:

  ```
  public class Customer
  {
  public int CustomerID { get;
  set; }
  public string Name { get; set; }
  // Navigation property
  public virtual ICollection<Order>
  Orders { get; set; }
  }
  public class Order
  {
  public int ID { get; set; }
  public string ProductName { get;
  set; }
  public int Price { get; set; }
  // Foreign key
  public int CustomerID { get;
  set; }
  // Navigation properties
  public virtual Customer cust
  { get; set; }
  }
  ```

  In the preceding code snippet, `Orders` is the navigational property in the `Customer` class and `cust` is the navigational property in the `Order` class. We call these as navigational properties because they allow us to navigate to the properties of another class. For example, by using the navigational property, `cust`, you can navigate to the orders associated with that customer. One customer can place multiple orders. Therefore, the orders navigational property is declared as a collection. This indicates a one-to-many relationship between the `Customer` and `Order` classes.

  In addition, a property by the name, `CustomerID`, is included in the `Order` class. This property is inferred by Entity Framework as a foreign key.

  Entity Framework uses certain conventions to identify a foreign key. Entity Framework infers that a property is a foreign key, if it meets the following conditions:
- The property has the same data type as the principal primary key. For example, the

`CustomerID` property in the `Order` class has the same data type as the principal primary key, `CustomerID`, in the `Customer` class.

- ❑ The name of the property follows any one of the following formats:
  - **<navigational property name><principal primary key property name>**: As per this format, a valid name for the foreign key in the `Order` class could be `custCustomerID`.
  - **<principal class name><principal primary key property name>:** As per this format, a valid name for the foreign key in the `Order` class could be `CustomerCustomerID`.
  - **<principal primary key property name>**: As per this format, a valid name for the foreign key in the `Order` class could be `CustomerID`.

If there are multiple properties that match the given formats, the precedence is given in the order the preceding formats are listed.

Using the Code-first Approach

You can use the code-first approach by performing the following steps:

1. Create the model classes.
2. Create the database context.
3. Configure the database location.
4. Create a controller.

Creating the Model Classes

You can create a model by simply using C# classes. For example, you can create the following class to describe the data related to customers:

```
public class Customer
{
 public int ID { get; set; }
 public string Name { get; set; }
 public string Gender { get; set; }
 public string Address { get; set; }
 public string Email { get; set; }
}
```

In the preceding code snippet, the class named `Customer` is created. The `Customer` class defines various properties, such as `ID`, `Name`, `Gender`, `Address`, and `Email`.

In addition, the following class can be created to describe the data related to products:

```
public class Product
{
 public int ID { get; set; }
 public string Name { get; set; }
 public string Description { get;
 set; }
}
```

In the preceding code snippet, the class named `Product` is created. The `Product` class defines various properties, such as `ID`, `Name`, and `Description`.

Creating the Database Context

Once you have created the model, you need to define the database context class. This context class coordinates with Entity Framework and allows you to query and save the data in the database. You can define the database context class by deriving from the `System.Data.Entity.DbContext` class. The database context class exposes one or more properties of the type, `DbSet <T>`, where `T` represents the type of object that needs to be stored in the database.

For example, consider the following code snippet:

```
public class ShopDataContext :
DbContext
 {
 public DbSet<Customer> Customers
 { get; set; }
 public DbSet<Product> Products { get;
 set; }
 }
```

In the preceding code snippet, the database context class named `ShopDataContext` is created. It is derived from the `DBContext` class defined in the `System.Data.Entity` namespace. This class creates the `DBSet` property for both, the `Customer` class and the `Product` class.

Configuring the Database Location

Entity Framework automatically creates the database in SQL Server with the name of the database context. However, it is advisable to configure the database location to keep track of the database that is being used. To configure the database location, you need to define a connection string in the **Web.config** file. A connection string specifies information about a data source and the way to establish a connection. It consists of a series of distinct pieces of information known as connection string properties. These properties are separated by semicolons in the code.

The connection string can have various connection string properties. Some of the important connection string properties are:

- ❑ `Data Source`: It indicates the name or the IP address of the server where the data source is located.
- ❑ `Initial Catalog`: It indicates the name of the database that will be accessed by using the specified connection.
- ❑ `Integrated Security`: You can connect to

SQL Server by using either the Windows authentication or the SQL Server authentication. The `Integrated Security` property indicates that you want to connect to SQL Server by using the Windows user account. The possible values for this property are `True`, `False`, and `SSPI`. The True and SSPI (Security Support Provider Interface) values specify that the current Windows account credentials need to be used for user's authentication. However, `SSPI` provides better security. The `False` value specifies that the SQL Server authentication needs to be used.

- ❑ `Persist Security Info`: It indicates whether the user-related information, such as the user ID and password, is discarded once used to open the connection. This information is discarded when the value of `Persist Security Info` is set to `False`. However, if the value of this attribute is set to True, the security-sensitive information, including the user ID and password, can be obtained from the connection after the connection has been opened. This may lead to serious security issues as the login information can be leaked out as long as the connection is open.
- ❑ `AttachDBFilename`: It gets or sets a string that contains the name of the primary data file.
- ❑ `providerName`: It is an optional attribute that specifies a class to communicate with a specific type of data source.

To define a connection string, you need to open the **Web.config** file and add a new connection string within the `<connectionStrings>` and `</connectionStrings>` tags, as shown in the following code snippet:

```
<connectionStrings>
...
...
<add name="ShopDBConnection"
connectionString="Data Source=(LocalDb)
\v11.0;Initial Catalog=aspnet-
NewbayOnline-20130416063703;Integrated
Security=SSPI;AttachDBFilename=|
DataDirectory|\aspnet-
NewbayOnline-20130416063703.mdf"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

The preceding code snippet specifies the connection string named `ShopDBConnection`. This connection string specifies that the data source, `(LocalDb)\v11.0`, and the SQL Server database named **aspnet-NewbayOnline-20130416063703.mdf** need to be used to store the data.

Afterwards, you can specify the connection string to be used by adding a constructor in the `DbContext` class. For example, consider the following code snippet:

```
public class ShopDataContext:DbContext
 {
 public ShopDataContext()
 : base("ShopDBConnection")
 {
 }
 }
```

In the preceding code snippet, the connection string named `ShopDBConnection` is specified by using the constructor in the `ShopDataContext()` class. As a result, a connection to the database will be established by using the `ShopDBConnection` connection string.

Creating a Controller

You can create a controller either by using scaffolding or by writing the code manually. For creating the controller automatically by using scaffolding, you need to specify the scaffolding template, model class name, and data context name. One of the scaffolding templates creates an MVC controller with the various action methods and the corresponding views. These action methods handle the create, edit, display, and delete operations in a database. However, you can also create these methods manually in an empty controller according to your needs. Let us discuss these action methods one by one.

The Index Action Method

The `Index()` action method handles the operation of retrieving the model class objects from a data store and rendering a view to list the model class objects.

Consider the following `Index()` action method that retrieves customer records and renders an appropriate view:

```
public ActionResult Index()
{
 var db = new ShopDataContext();
 var customers = db.Customers;
 return View(customers);
}
```

In the preceding action method, first the instance, `db`, of the `ShopDataContext` class is created. Next, this instance is used to retrieve the customer records and store them in the collection variable, `customers`. Finally, the `View()` method is invoked to render a view and the `customers` collection is passed to the view as a parameter.

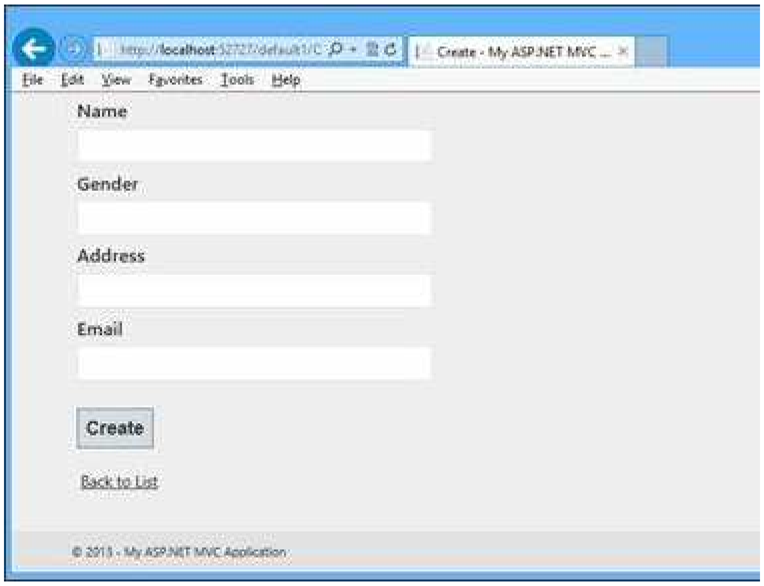
NOTE *The preceding `Index()` method uses Language Integrated Query (LINQ) to retrieve customer records from the database. You will learn more about LINQ in the next chapter.*

The Create Action Method

The Create() action method handles the operation for creating model class objects. The scaffolding template creates two Create() action methods in a controller. One of the Create() action methods is invoked whenever the HTTP GET request is received. This action method renders the Create view. The other Create() action method is invoked whenever the HTTP POST request is received from the Create view. This action method first validates the data entered by the user in the Create view, and then saves the data in the database. For example, suppose you have created a controller for the Customer model, then the following Create() action method is created in the controller:

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

The preceding method returns a view that displays a form to enter customer records, as shown in the following figure.


http://localhost:52727/default1/C
Create - My ASP.NET MVC ...
File
Edit
View
Favorites
Tools
Help
Name
Gender
Address
Email
Create
Back to List
© 2013 - My ASP.NET MVC Application

A Form to Enter Customer Records

In addition, the following Create() method is also created in the controller:

```
[HttpPost]
public ActionResult Create(Customer
customer)
 {
 if (ModelState.IsValid)
 {
 db.Customers.Add(customer);
 db.SaveChanges();
 return RedirectToAction("Index");
 }
 return View(customer);
 }
```

In the preceding method, the [HttPPost] attribute is included, which restricts an action method to handle only HTTP POST requests. The Create () action method accepts the customer object containing the data entered by the user as a parameter. If the data is valid, an instance of the appropriate DbContext class (db) is used to invoke the Add() method that adds the customer object to the Customers property of the data context. Then, the SaveChanges() method is used to save the data related to the customer object in the database. Finally, the user is redirected to the Index view.

The Edit Action method

The Edit () action method handles the operation of editing the data related to a particular model object. The scaffolding template adds two edit action methods related to a model object. One of the edit action methods finds the data related to a customer whose records need to be edited and renders the Edit view. For example, if you have created the Customer class in the model, the following Edit() method in the controller is created that enables you to return a view for a particular customer:

```
public ActionResult Edit(int id = 0)
{
 Customer customer = db.Customers.Find
 (id);
 if (customer == null)
 {
 return HttpNotFound();
 }
 return View(customer);
}
```

The preceding action method accepts the id of the record to be edited as a parameter. It then uses the Find() method to find the customer record on the basis of the primary key, id. If the customer record is found, the view for editing the customer record is rendered. Otherwise, the HttpNotFound() method is invoked, which informs you that the requested resource is not found.

The other Edit () action method is invoked whenever the HTTP POST request is received from the Edit view. This action method first validates the data entered by the user, and then saves the updated data. For example, consider the following method:

```
[HttpPost]
public ActionResult Edit(Customer
customer)
{
 if (ModelState.IsValid)
 {
 db.Entry(customer).State =
 EntityState.Modified;
 db.SaveChanges();
 return RedirectToAction("Index");
 }
 return View(customer);
}
```

In the preceding method, the `ModelState.IsValid` property validates if the data submitted in the form is valid. If the data is valid, the state of the entity is set to modified. This state marks the state of an entity as modified. Further, all the property values will be saved to the database by calling the `SaveChanges()` method of the database context. After saving the data, the code redirects the user to the `Index ()` action method of the controller, which displays all the `customer` records, including the changes just made.

The Details Action method

The `Details()` action method contains the code to find the details related to a particular model object. It finds the records on the basis of the parameter value passed to it. The parameter is the primary key, such as `id`. For example, consider the following code to retrieve the data related to a particular customer:

```
public ActionResult Details(int id = 0)
 {
 Customer customer = db.Customers.Find
 (id);
 if (customer == null)
 {
  return HttpNotFound();
  }
  return View(customer);
 }
```

In the preceding code, the `Find()` method is used to find the customer record on the basis of the primary key, `id`. If the customer record is found, the view containing the customer record is rendered. Otherwise, the `HttpNotFound()` method is invoked, which informs you that the requested resource is not found.

The Delete Action method

The `Delete()` action method handles the operation of deleting the data related to a particular model object. The scaffolding template adds two `Delete ()` action methods related to a model object, one for the HTTP GET request and the other for the HTTP POST request. The HTTP GET version of the `Delete()` action method finds the data related to a customer whose record needs to be deleted and renders a view that displays the searched record and provides an option to delete the record. For example, if you have created the `Customer` class in the model, the following `Delete()` method in the controller is created that enables you to return a view for a particular customer:

```
public ActionResult Delete(int id = 0)
 {
 Customer customer = db.Customers.Find
 (id);
 if (customer == null)
 {
 return HttpNotFound();
 }
 return View(customer);
 }
```

The HTTP POST version of the `Delete()` action method requires the same signature as that of the HTTP GET version of the `Delete()` action method. However, CLR requires overloaded methods to have unique signatures. To resolve this problem, the HTTP POST method is named as `DeleteConfirmed`. The `DeleteConfirmed()` action method deletes the required record, and then saves the changes in the database. For example, consider the following code snippet:

```
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int
id)
{
 Customer customer = db.Customers.Find
 (id);
 db.Customers.Remove(customer);
 db.SaveChanges();
 return RedirectToAction("Index");
}
```

The preceding `DeleteConfirmed()` action method is preceded with the `ActionName("Delete")` attribute. This attribute is added to perform mapping for the routing system so that a URL that includes /Delete/ for a POST request will be routed to the `DeleteConfirmed()` action method. In the `DeleteConfirmed()` action method, the `Remove()` method is used to delete a customer record and the `SaveChanges()` method is used to save the changes to the database. Finally, the code redirects the user to the `Index()` action method of the controller. The `Index()` action method displays the customers' records, which exclude the deleted record.

Task5.1: Creating a Controller by using Scaffolding

Activity 5.1: Working with the EF Code-first Approach

Initializing a Database with Test Data

During the development of a Web application, several changes may be made to the model to implement various features. As a result, the model and the previously-created database may not remain in sync with each other. To maintain synchronization between the model and the database, you need to ensure that any change in the model is reflected back in the database. Therefore, the database needs to be recreated. For this, Entity Framework provides the following two classes in the `System.Data.Entity` namespace:

- **DropCreateDatabaseAlways:** It allows you to recreate an existing database whenever the application starts.
- **DropCreateDatabaseIfModelChanges:** It allows you to recreate an existing database whenever the model changes.

Depending on the need, you can select one of the preceding classes while calling the `SetInitializer ()` method of the `System.Data.Entity.Database` namespace. For example, if you want to recreate the NewBay store database whenever the application starts, you need to write the highlighted portion of the following code inside the **Global.asax.cs** file:

```
protected void Application_Start()
{
Database.SetInitializer(new
DropCreateDatabaseAlways<ShopDataConte
xt>());
AreaRegistration.RegisterAllAreas();
RegisterGlobalFilters
(GlobalFilters.Filters);
RegisterRoutes(RouteTable.Routes);
}
```

In the preceding code, the `DropCreateDatabaseAlways` class is used while calling the `SetInitializer()` method. This ensures that the existing database is recreated whenever the application starts.

However, if you want to recreate the NewBay store database only when the model changes, you need to use the highlighted portion of the following code inside the **Global.asax.cs** file:

```
protected void Application_Start()
{
Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<ShopD
ataContext>());
AreaRegistration.RegisterAllAreas();
RegisterGlobalFilters
(GlobalFilters.Filters);
RegisterRoutes(RouteTable.Routes);
}
```

In the preceding code, the `DropCreateDatabaseIfModelChanges` class is used while calling the `SetInitializer()` method. This ensures that the existing database is recreated whenever the model changes.

At the time of development, you can instruct Entity Framework to initialize your database with some sample data for testing purposes whenever the database is recreated. For this, you need to create a model class that derives from either the `DropCreateDatabaseIfModelChanges` class or the `DropCreateDatabaseAlways` class. In this class, you need to override the `Seed()` method. The `Seed()` method enables you to define the initial data for the application.

Consider the following code snippet that defines the`MyDataContextDbInitializer` class:

```
public                                   class
MyDataContextDbInitializer :DropCreateD
atabaseIfModelChanges<ShopDataContext>
{
protected override void Seed
(ShopDataContext context)
{
context.Customers.Add(new Customer()
{ Name = "Test Customer",
Gender="Male",Email="test@yahoo.com" }
);
base.Seed(context);
}
}
```

In the preceding code snippet, the `MyDataContextDbInitializer` class in derived from the `DropCreateDatabaseIfModelChanges` class. In the `MyDataContextDbInitializer` class, the `Seed()` method is overridden to define the initial data for the customers.

Then, you need to register the `MyDataContextDbInitializer` class while calling the `SetInitializer()` method. For this, you need to add the highlighted portion of the following code snippet in the `Application_Start()` method in the **Global.asax.cs** file:

```
protected void Application_Start()
{
System.Data.Entity.Database.SetInitial
izer(new MyDataContextDbInitializer
());
}
```

You need to ensure that the required namespaces are included. Otherwise, the code will not run.

Entity Framework Code-first Migrations

The approach of recreating a database whenever the application starts or the model changes will lead to the deletion of existing data. Therefore, this approach is applicable only in the development phase when you are working with some sample data. However, when the Web application becomes live, you need a mechanism that enables you to prevent the loss of data. For this, `Entity Framework` supports code-first migrations. It allows you to change the model and modifies the database accordingly. For example, you have created the Customer model with the properties, such as `Name`, `Address`, and `Gender`. Now, you want to add additional properties named `City` and `PhoneNumber` in the `Customer` model. A database migration allows you to add the column in the database without dropping the database, thereby retaining the data saved in the database. To enable database migration, you need to use Package Manager Console. You can use Package Manager Console to:

- ❑ Enable migrations
- ❑ Run migrations

Enabling Migrations

To enable migrations, you need to run the following command in Package Manager Console:

```
Enable-Migrations
```

If more than one context is used in the application, you need to specify the context name. For example, consider the following command:

```
Enable-Migrations          -ContextTypeName
NewbayOnline.Models.ShopDataContext
```

In the preceding code snippet, the `ShopDataContext` class is specified as the context name while running the `Enable-Migrations` command.

On execution, this command adds the `Migration` folder to the project. This `Migration` folder contains the following two files:

- ❑ **The Configuration class:** This class enables you to configure how the migrations will behave for the data context. It contains the `Seed()` method that can be modified according to your needs. To use the `Configuration` class and modify the `Seed()` method, you first need to set the `AutomaticMigrationsEnabled` variable to `true` in the `Configuration()` method of the Configuration class file, as shown in the following code snippet:

```
public Configuration()
 {
 AutomaticMigrationsEnabled =
 true;
 }
```

Thereafter, you can modify the `Seed()` method in the Configuration class file, as shown in the following code snippet:

```
protected   override   void   Seed
(NewBayOnline.Models.ShopDataCont
ext context)
 {
 context.Customers.AddOrUpdate(r
 => r.Name, new Customer { Name =
 "abc", Gender = "Male" });
 }
```

The preceding code snippet will add the customer record to the database only if the record does not already exist. However, if the record related to that customer already exists, it will be updated if there is any modification in the record.

NOTE *If you have registered the model class for initializing the database with the test data, you need to comment the `SetInitializer` statement in the `Application_Start()` method of the* ***Global.asax.cs*** *file.*

- ❑ **An InitialCreate migration script:** This migration script is only added in the project when you already have a database, before you enable migrations. It contains two methods, `Up()` and `Down()`. The `Up()` method contains the code to migrate the database to your new model, and the `Down()` method contains the code to undo the change and revert back to the previous version.

Running Migrations

You can run migration by using the `Update-Database` command. This command updates the database by applying the pending migrations. For example, if you have added a new property in the model, the following command can be used to update the database:

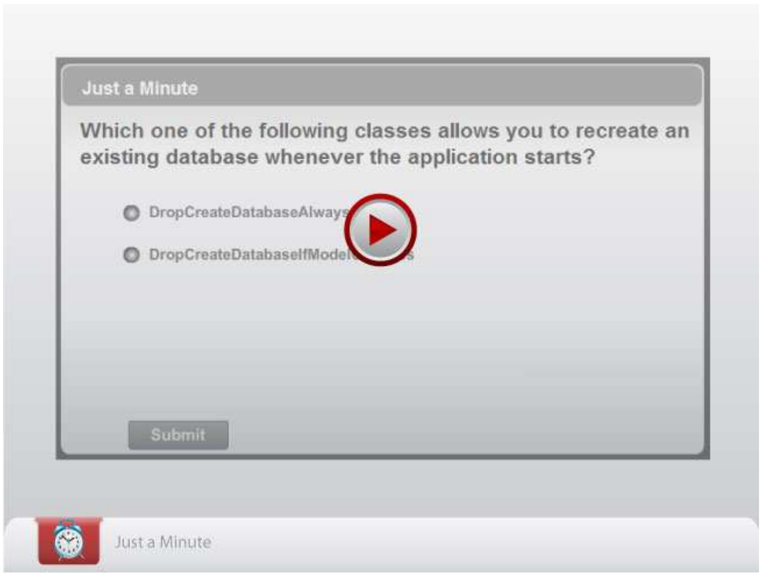
```
Update-Database -Verbose
```

In the preceding command, the `-Verbose` flag is used to view the SQL statements that are being applied to the target database.

Activity 5.2: Managing Changes to the Model

Task 5.2: Working with the EF Database-first Approach

Task 5.3: Working with the EF Model-first Approach


Just a Minute

Which one of the following classes allows you to recreate an existing database whenever the application starts?

- DropCreateDatabaseAlways
- DropCreateDatabaseIfModel s

Summary

In this chapter, you learned that:

- ❑ Entity Framework enables you to work with relational data in the form of objects by representing the relational objects, such as tables, views, stored procedures, and functions, in the form of a conceptual model in the application.
- ❑ While developing an application, if you already have an existing database, the database-first approach is considered to be the best-suited approach.
- ❑ In the model-first approach, the first priority is the model, while the code and the database are secondary.
- ❑ In the code-first approach, you can start the development of your application by coding custom classes and properties, which correspond to tables and columns in the relational database.
- ❑ A relationship between two entities can be defined by using navigational properties.
- ❑ You can use the code-first approach by performing the following steps:
  a. Create the model classes.
  b. Create the database context.
  c. Configure the database location.
  d. Create a controller.
- ❑ A connection string specifies information about a data source and the way to establish a connection.
- ❑ Entity Framework provides the following two classes in the `System.Data.Entity` namespace:
  - `DropCreateDatabaseAlways`
  - `DropCreateDatabaseIfModelChanges`
- ❑ A database migration allows you to add the column in the database without dropping the database, thereby retaining the data saved in the database.
- ❑ You can use Package Manager Console to:
  - Enable migrations
  - Run migrations

Glossary

B

Bundling

Bundling is a technique provided by ASP.NET MVC that allows you to combine multiple files, such as CSS and JavaScript, into a single file.

Business logic layer

Consists of the components of the application that control the flow of execution and communication between the presentation and data layers.

C

Client-side scripting

Client-side scripting enables you to develop Web pages that can dynamically respond to user input without having to interact with a Web server.

Controller

Refers to a set of classes that handle communication from the user and the overall application flow.

Cookies

Cookies are small pieces of information that are stored on the client&#8217;s computer.

D

Data layer

Consists of components that expose the application data stored in databases to the business logic layer

Dynamic Web page

A Web page whose content is generated dynamically by a Web application or that responds to user input and provides interactivity is called a dynamic Web page.

F

Fat client and thin server

The architecture in which the business logic layer resides on the client tier is referred to as the fat client and thin server architecture. In this architecture, the client accepts user requests and processes these requests on its own.

Fat server and thin client

The architecture in which the business logic layer resides on the server is referred to as the fat server and thin client architecture. In this architecture, the client accepts requests from the users and forwards the same to the server.

J

JavaScript library

AJavaScript library is a set of prewritten JavaScript code, which helps in developing a JavaScript-based application easily.

jQuery

jQuery is a cross-browser JavaScript library that helps you easily perform various tasks, such as DOM traversal, event handling, and animating elements.

jQuery UI

jQuery UI is an organised set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript library.

L

LINQ

LINQ offers a consistent programming model to query data from different data sources.

M

Model

Refers to a set of classes that describes the data that the application works with.

N

Nested layout

A nested layout page refers to a layout that is derived from a parent layout.

P

Presentation layer

Consists of the interface through which the users interact with the application.

R

Razor

Razor is a markup syntax that allows you to embed server-side code (written in C# or VB) in an HTML markup.

Routing

Routing is a feature that enables you to develop applications with comprehensible and searchable URLs.

S

Scaffolding

MVC provides a scaffolding feature that provides a quick way to generate the code for commonly used operations in a standardized way.

Server-side scripting

Server-side scripting provides users dynamic content that is based on the information stored at a remote location, such as a back-end database.

Static Web Page

A Web page that contains only static content and is delivered to the user as it is stored is called a static Web page.

U

Unobtrusive JavaScript

Unobtrusive JavaScript is a general approach to implement JavaScript in Web pages. In this approach, the JavaScript code is separated from the HTML markup.

V

View

Refers to the components that define an application's user interface.

ViewBag

ViewBag is a dynamic object that allows passing data between a controller and a view.

ViewData

ViewData is a dictionary of objects derived from the ViewDataDictionary class.