ASP.NET MVC

Mentor as a Service

ASP.NET

ASP.NET is used to develop database-driven, highly-functional, and scalable dynamic websites that use client-side and server-side code. You can create many kinds of website with ASP.NET such as web portals, online shopping sites, blogs, and wikis.

Programming Models

When you use ASP.NET we can choose different programming models.

• **Web Pages:**

When you build a site by using Web Pages, you can write code by using the C# or Visual Basic programming language. If you write C# code, these pages have a .cshtml file extension. If you write Visual Basic code, these pages have a .vbhtml file extension. ASP.NET runs the code in these pages on the server to render data from a database, respond to a form post, or perform other actions. This programming model is simple and easy to learn and is suited for simple data-driven sites. ASP.NET includes Web Pages.

• **Web Forms:**

When you build a site by using Web Forms, you employ a programming model with rich server-side controls and a page life cycle that is not unlike building desktop applications. Built-in controls include buttons, text boxes, and grid views for displaying tabulated data. You can also add third-party controls or build custom controls. To respond to user actions, you can attach event handlers containing code to the server-side controls. For example, to respond to a click on a button called firstButton, you could write code in the firstButton_Click() event handler.

• **MVC:**

When you build a site by using ASP.NET MVC, you separate server-side code into three parts:

Model.

An MVC model defines a set of classes that represent the object types that the web application manages. For example, the model for an ecommerce site might include a Product model class that defines properties such as Description, Catalog Number, Price, and others. Models often include data access logic that reads data from a database and writes data to that database.

Controllers.

An MVC controller is a class that handles user interaction, creates and modifies model classes, and selects appropriate views. For example, when a user requests full details about a product, the controller creates a new instance of the Product model class and passes it to the Details view, which displays it to the user.

Views.

An MVC view is a component that builds the webpages that make up the web application's user interface. Controllers often pass an instance of a model class to a view. The view displays properties of the model class. For example, if the controller passes a Product object, the view might display the name of the product, a picture, and the price. This separation of model, view, and controller code ensures that MVC applications have a logical structure, even for the most complex sites. It also improves the testability of the application. Ultimately, ASP.NET MVC enables more control over the generated HTML than either Web Pages or Web Forms.

The WEB API

Whichever programming model you choose, you have access to classes from the ASP.NET API. These classes are included in the .NET Framework in namespaces within the System.Web namespace and can be used to rapidly implement common website functionalities such as:

configuration.

Using Web.config files, you can configure your web application, regardless of the programming model. Web.config files are XML files with specific tags and attributes that the ASP.NET runtime accepts. For example, you can configure database connections and custom error pages in the Web.config file. In code, you can access the configuration through the System.Web.Configuration namespace.

Authentication and Authorization.

Many websites require users to log on by entering a user name and password, or by providing extra information. You can use ASP.NET membership providers to authenticate and authorize users and restrict access to content. You can also build pages that enable users to register a new account, reset a password, recover a lost password, or perform other account management tasks. Membership providers belong to the System.Web.Security namespace.

Caching.

ASP.NET may take some time to render a complex webpage that may require multiple database queries or calls to external web services. You can use caching to mitigate this delay. ASP.NET caches a rendered page in memory, so that it can return the same page to subsequent user requests without having to render it again from the start. In a similar manner, .NET Framework objects can also be cached. You can access cached pages by using the System.Runtime.Caching namespace and configure the caches in Web.config.

Compiling ASP.NET Code

Because ASP.NET server-side code uses the .NET Framework, you must write code in a .NET managed programming language such as C# or Visual Basic. Before running the code, it must be compiled into native code so that the server CPU can process it. This is a two-stage process:

1. **Compilation to MSIL.** When you build a website in Visual Studio, the ASP.NET compiler creates .dll files with all the code compiled into Microsoft Intermediate Language (MSIL). This code is both independent of the language you used to write the application and the CPU architecture of the server.

2. Compilation to native code. When a page is requested for the first time, the Common Language Runtime (CLR) compiles MSIL into native code for the server CPU.

This two-stage compilation process enables components written in different languages to work together and enables many errors to be detected at build time. Note, however, that pages may take extra time to render the first time they are requested after a server restart. To avoid this delay, you can pre-compile the website.

When you use the default compilation model, delays can arise when the first user requests a page. This is because ASP.NET must compile the page before serving it to the browser. To avoid such delays and to protect source code, use pre-compilation. When you pre-compile a site, all the ASP.NET files, including controllers, views, and models, are compiled into a single .dll file.

Client-Side Web Technologies

Originally, in ASP.NET, and similar technologies like PHP, all the code ran on the web server. This approach is often practical because the web server usually has immediate access to the database and more processor power and memory than a typical client computer. However, in such an approach, every user action requires a round trip between the client and the web server, and most actions require a complete reload of the page. This can take significant time. To respond quickly and provide better user experience, you can supplement server-side code with client-side code that runs in the web browser.

JavaScript

JavaScript is a simple scripting language that has syntax like C#, and it is supported by most web browsers. A scripting language is not compiled. Instead, a script engine interprets the code at run time so that the web browser can run the code.

Javascript is supported by virtually every web browser. can include JavaScript code in your ASP.NET pages, irrespective of the programming model you choose. JavaScript is a powerful language but can require many lines of code to achieve visual effects or call external services. Script libraries contain pre-built JavaScript functions that help implement common actions that you might want to perform on client-side code. You can use a script library, instead of building all your own JavaScript code from the start; using a script library helps reduce development time and effort. Different browsers interpret JavaScript differently. When you develop an Internet site, you do not know what browsers site visitors use. Therefore, you must write JavaScript that works around browser compatibility.

jQuery

jQuery is one of the most popular JavaScript libraries. It provides elegant functions for interacting with the HTML elements on your page and with cascading style sheet (CSS) styles. For example, you can locate all the elements on a webpage and change their background color by using a single line of code. To achieve the same result by using JavaScript only, you need to write several lines of code and a programming loop. Furthermore, the code you write may be different for different browsers.

Using jQuery, it is easier to write code to respond to user actions and to create simple animations. jQuery also handles browser differences. You can use jQuery to call web services on remote computers and update the webpage with the results returned.

There are JavaScript frameworks are available for building Single Page Applications such as **Angular**, **React js**, **Vue js**, etc. It is another learning curve for understanding JavaScript Client-side Frameworks for building Client-Side User Interface for Web Applications.

AJAX

AJAX is a technology that enables browsers to communicate with web servers asynchronously by using the XmlHttpRequest object without completely refreshing the page. You can use AJAX in a page to update a portion of the page with new data, without reloading the entire page. For example, you can use AJAX to obtain the latest comments on a product, without refreshing the entire product page. AJAX is an abbreviation of Asynchronous JavaScript and XML. AJAX is implemented entirely in JavaScript, and ASP.NET, by default, relies on the jQuery library to manage AJAX requests and responses. The code is run asynchronously, which means that the web browser does not freeze while it waits for an AJAX response from the server. Initially, developers used XML to format the data returned from the server. More recently, however, developers use JavaScript Object.

Internet Information Server

Every website must be hosted by a web server. A web server receives requests for web content from browsers, runs any server-side code, and returns webpages, images, and other content. HTTP is used for communication between the web browser and the web server. Internet Information Server is a web server that can scale up from a small website running on a single web server to a large website running on a multiple web server farms. Internet Information Server is available with Windows Server.

Internet Information Server Features

Internet Information Server is tightly integrated with ASP.NET and Windows Server. It includes the following features:

1. **ASP.NET Support.** IIS is a web server that fully supports ASP.NET.
2. **Authentication and Security.** IIS supports most common standards for authentication, including Smart Card authentication and Integrated Windows authentication. You can also use Secure Sockets Layer (SSL) to encrypt security-sensitive communications, such as logon pages and pages containing credit card numbers.
3. **High Performance Caches.** You can configure ASP.NET to make optimal use of the IIS caches to accelerate responses to user requests. When IIS serves a page or other content, it can cache it in memory so that subsequent identical requests can be served faster.
4. **Centralized Web Farm Management.** When you run a large website, you can configure a load balanced farm of many IIS servers to scale to large sizes. IIS management tools make it easy to deploy sites to all servers in the farm and manage sites after deployment.
5. **Deployment Protocols.** The advanced Web Deploy protocol, which is built into Visual Studio, automatically manages the deployment of a website with all its dependencies. Alternatively, you can use File Transfer Protocol (FTP) to deploy content.
6. **Other Server-Side Technologies.** You can host websites developed in PHP and Node.js on IIS.

Scaling Up IIS

A single web server has limited scalability because it is limited by its processor speed, memory, disk speed, and other factors. Furthermore, single web servers are vulnerable to hardware failures and outages. For example, when a single web server is offline, your website is unavailable to users. You can improve the scalability and resilience of your website by hosting it on a multiple server farm. In such server farms, many servers share the same server name. Therefore, all servers can respond to browser requests. A load balancing system such as Windows Network Load Balancing or a hardware-based system

such as Riverbed Cascade, distributes requests evenly among the servers in the server farm. If a server fails, other servers are available to respond to requests, and thereby, the website availability is not interrupted. IIS servers are designed to work in such server farms and include advanced management tools for deploying sites and managing member servers. Perimeter Networks Web servers, including IIS, are often located on a perimeter network. A perimeter network has a network segment that is protected from the Internet through a firewall that validates and permits incoming HTTP requests. A second firewall, which permits requests only from the web server, separates the perimeter network from the internal organizational network. Supporting servers, such as database servers, are usually located on the internal organizational network.

In this configuration, Internet users can reach the IIS server to request pages and the IIS server can reach the database server to run queries. However, Internet users cannot access the database server or any other internal computer directly. This prevents malicious users from running attacks and ensures a high level of security.

IIS Express

Internet Information Server Express does not provide all the features of Internet Information Server on Windows Server. For example, you cannot create load-balanced server farms by using Internet Information Server Express. However, it has all the features necessary to host rich ASP.NET websites and other websites on a single server.

Other Web Servers

Apache is a popular non-Microsoft web server and there are other alternatives such as nginx. Apache can be installed on Windows Server or Windows client computers to host websites during development or for production deployments.

ASP.NET MVC 5

Models, Views, and Controllers

Models represent data and the accompanying business logic, controllers interact with user requests and implement input logic, and views build the user interface. By examining how a user request is processed by ASP.NET MVC 5, you can understand how the data flows through models, views, and controllers before being sent back to the browser.

Models and Data

A model is a set of .NET class that represents objects handled by your website. For example, the model for an e-commerce application may include a Product model class with properties such as Product ID, Part Number, Catalog Number, Name,Description, and Price. Like any .NET class, model classes can include a constructor, which is a procedure that runs when a new instance of that class is created. You can also include other procedures, if necessary. These procedures encapsulate the business logic. For example, you can write a Publish procedure that marks the Product as ready-to-sell. Most websites store information in a database. In an MVC application, the model includes code that reads and writes database records. ASP.NET MVC works with many different data access frameworks. However, the most commonly used framework is the Entity Framework.

Controllers and Actions

A controller is a .NET class that responds to web browser requests in an MVC application. There is usually one controller class for each model class. Controllers include actions, which are methods that run in response to a user request. For example, the Product Controller may include a Purchase action that runs when the user clicks the Add To Cart button in your web application. Controllers inherit from the System.Web.Mvc.Controller base class. Actions usually return a System.Web.Mvc.ActionResult object.

Views and Razor

A view is, by default, a .cshtml or .vbhtml file that includes both HTML markup and programming code. A view engine interprets view files, runs the server-side code, and renders HTML to the web browser. Razor is the default view engine in ASP.NET MVC 5.

The following lines of code are part of an ASP.NET MVC 5 view and use the Razor syntax. The @ symbol delimits server-side code.

```
<h2>Details</h2>
 <fieldset>
  <legend>Comment</legend>
  <div class="display-label">
       @Html.DisplayNameFor(model => model.Subject)
  </div>
  <div class="display-field">
      @Html.DisplayFor(model => model.Subject)
  </div>
  <div class="display-label">
      @Html.DisplayNameFor(model => model.Body)
  </div>
  <div class="display-field">
      @Html.DisplayFor(model => model.Body)
  </div>
</fieldset>
```

Often, the view displays properties of a model class. In the preceding code example, the Subject property and Body property are incorporated into the page.

Request Life Cycle

The Request life cycle comprises a series of events that happen when a web request is processed. The following steps illustrate the process that MVC applications follow to respond to a typical user request. The request is for the details of a product with the ID "1":

1. The user requests the web address: http://localhost:4565/product/display/1
2. The MVC routing engine examines the request and determines that it should forward the request to the Product Controller and the Display action.
3. The Display action in the Product Controller creates a new instance of the Product model class.
4. The Product model class queries the database for information about the product with ID "1".
5. The Display action also creates a new instance of the Product Display View and passes the Product Model to it.
6. The Razor view engine runs the server-side code in the Product Display View to render HTML. In this case, the server-side code inserts properties such as Title, Description, Catalog Number, and Price into the HTML.
7. The completed HTML page is returned to the browser for display.

ASP.NET MVC State Management

In application development, the application state refers to the values and information that are maintained across multiple operations. Hypertext Transfer Protocol (HTTP) is fundamentally a stateless protocol, which indicates that it has no mechanism to retain state information across multiple page requests. However, there are many scenarios, such as the following, which require
state to be preserved:

• **User preferences.** Some websites enable users to specify preferences. For example, a photo sharing web application might enable users to choose a preferred size for photos. If this preference information is lost between page requests, users must continually reapply the preference.

• **User identity.** Some sites authenticate users to provide access to members-only content. If the user identity is lost between page requests, the user must re-enter the credentials for every page.

• **Shopping carts.** If the content of a shopping cart is lost between page requests, the customer cannot buy anything from your web application. In almost all web applications, state storage is a fundamental requirement. ASP.NET provides several locations where you can store state information, and simple ways to access the state information. However, you must plan the use of these mechanisms carefully. If you use the wrong location, you may not be able to retrieve a value when you expect to. Furthermore, poor planning of state management frequently results in poor performance.

In general, you should be careful about maintaining large quantities of state data because it either consumes server memory, if it is stored on the server, or slows down the transfer of the webpage to the browser, if it is included in a webpage. If you need to store state values, you can choose between client side state storage or server-side state storage.

Client-Side State Storage

When you store state information on the client, you ensure that server resources are not used. However, you should consider that all client-side state information must be sent between the web server and the web browser, and this process can slow down page load time. Use client-side state storage only for small amounts of data:

• **Cookies.** Cookies are small text files that you can pass to the browser to store information. A cookie can be stored:

- In the client computer memory, in which case, it preserves information only for a single user session.
- On the client computer hard disk drive, in which case, it preserves information across multiple sessions.

Most browsers can store cookies only up to 4,096 bytes and permit only 20 cookies per website. Therefore, cookies can be used only for small quantities of data. Also, some users may disable cookies for privacy purposes, so you should not rely on cookies for critical functions.

• **Query strings.** A query string is the part of the URL after the question mark and is often used to communicate form values and other data to the server. You can use the query string to preserve a small amount of data from one page request to another. All browsers support query strings, but some impose a limit of 2,083 characters on the URL length. You should not place any sensitive information in query strings because it is visible to the user, anyone observing the session, or anyone monitoring web traffic.

Server-Side State Storage

State information that is stored on the server consumes server resources, so you must be careful not to overuse server-side state storage or risk poor performance. The following locations store state information in server memory:

- **TempData.**
This is a state storage location that you can use in MVC applications to store values between one request and another. You can store values by adding them to the TempData collection. This information is preserved for a single request only and is designed to help maintain data across a webpage redirect. For example, you can use it to pass an error message to an error page.

- **Application State.**
This is a state storage location that you can use to store vales for the lifetime of the application. The values stored in application state are shared among all users. You can store values by adding them to the Application collection. If the web server or the web application is restarted, the values are destroyed. The Application_Start() procedure in the Global.asax file is an appropriate place to initialize application state values. Application state is not an appropriate place to store user specific values, such as preferences, because if you store a preference in application state, all users share the same preference, instead of having their own unique value.

- **Session state.**
The Session collection stores information for the lifetime of a single browser session and values stored here are specific to a single user session; they cannot be accessed by other users. By default, if the web server or the web application is restarted, the values are destroyed. However, you can configure ASP.NET to store session state in a database or state server. If you do this, session state can be preserved across restarts. Session state is available for both authenticated users and anonymous users. By default, session state uses cookies to identify users, but you can configure ASP.NET to store session state without using cookies.

If you choose to use these server memory locations, ensure that you estimate the total volume of state data that may be required for all the concurrent users that you expect to manage. Application state values are stored only once, but session state values are stored once for each concurrent user. Specify server hardware that can easily manage this load or move state data into the following server hard disk drive-based locations.

- **Profile properties.**
If your site uses an ASP.NET profile provider, you can store user preferences in profiles. Profile properties are persisted to the membership database, so they will be kept even if the web application or web server restarts.

- **Database tables.**
If your site uses an underlying database, like most sites do, you can store state information in its tables. This is a good place to store large volumes of state data that cannot be placed in server memory or on the client computer. For example, if you want to store a large volume of session-specific state information, you can store a simple ID value in the Session collection and use it to query and update a record in the database. Remember that state data is only one form of information that an ASP.NET application places in server memory. For example, caches must share memory with state data.

Search Engine Optimization (SEO)

Most users find web applications by using search engines. Users tend to visit the links that appear at the top of search engine results more frequently than those lower down and those on the second page of results. For this reason, website administrators and developers try to ensure their web application appears high in search engine results, by using a process known as Search Engine Optimization (SEO).

SEO ensures that more people visit your web application. Search engines examine the content of your web application, by crawling it with a program called a web bot. If you understand the priorities that web bots and search engine indexes use to order search results, you can create a web application that conforms to those priorities and thereby appears high in search engine results.

SEO Best Practices

Various search engines have different web bots with different algorithms to prioritize results. If you adopt the following best practices, your site has a good chance of appearing high in search results:

- Ensure that each webpage in your web application has a meaningful <title> element in the <head> section of the HTML.
- Ensure that you include a <meta name="keywords"> tag in the <head> element of each page. The content attribute of this element should include keywords that describe the content of the page accurately.
- Ensure that you include a <meta name="description"> tag in the <head> element of each page. The content attribute of this element should include a sentence that describes the content of the page accurately.
- Ensure that the <title> element and the <meta> elements are different for each page in your web application.
- Choose a domain name that includes one or more of your most important keywords.
- Ensure that keywords appear in the <h1>, <h2>, or <h3> elements of your webpage.
- Ensure that navigation controls enable web bots to crawl your entire web application. For example, you may have content in your site that users can only find with the search tool, not by clicking through links. As web bots cannot use search tools, this content will not be indexed.
- Ensure that URLs do not include GUIDs or long query text.

SEO and Web Application Structure

Information architecture is a subject that is closely related to SEO. This is because both information architecture and SEO are relevant to the structure, hierarchy, and accessibility of the objects in your web application. Users click links on menus to navigate to the pages that interest them. Web bots use the same links to navigate the web application and crawl its content. Users prefer URLs without GUIDs and long query text because they are meaningful. Web bots often ignore links with GUIDs and long query text in them. In addition, when keywords appear in URLs, web bots prioritize a webpage in search results. As an ASP.NET MVC developer, you must understand SEO principles and use them whenever you write code, to ensure that you do not damage the search engine positioning of your web application. Views are critical to SEO because they render <meta> tags, and <title> elements. Routes and the configuration of the routing engine are also critical, because, by using routes, you can control the URLs that your web application generates.

ASP.NET MVC Routing

ASP.NET enables developers to control the URLs that a web application uses, to link the URLs and the content by configuring routes. A route is an object that parses a requested URL, and it determines the controller and action to which the request must be forwarded. Such routes are called incoming routes.

HTML helpers also use routes when they formulate links to controllers and actions. Such routes are called outgoing routes.

You need to know how to write code that adds a new route to your application. You also need to know how the routing engine interprets a route so that requests go to the appropriate controllers, and users see meaningful URLs.

The ASP.NET Routing Engine

Routing governs the way URLs are formulated and how they correspond to controllers and actions. Routing does not operate on the protocol, server, domain, or port number of a URL, but only on the directories and file name in the relative URL.

For example, in the URL,

http://localhost:4565/photo/display/1

routing operates on the relative path /photo/display/1.

In ASP.NET, routes are used for two purposes:

- **To parse the URLs requested by browsers.**
  - This analysis ensures that requests are forwarded to the right controllers and actions.
    - These are called incoming URLs.

- **To formulate URLs in webpage links and other elements.**
  - When you use helpers such as Html.ActionLink() and Url.Action() in MVC views, the helpers construct URLs according to the routes in the routing table.
    - These are called outgoing URLs.

When you create an MVC web application in Visual Studio by using a project template, the application has a default route.

The Default Route

The default route is simple but logical and works well in many web applications. The default route examines the first three segments of the URL. Each segment is delimited by a forward slash:

- The first segment is interpreted as the **name of the controller**. The routing engine forwards the request to this controller. If a first segment is not specified, the default route forwards the request to a controller called Home.

- The second segment is interpreted as the **name of the action**. The routing engine forwards the request to this action. If a second segment is not specified, the default route forwards the request to a controller called Index.

- The third segment is interpreted as an **ID value**, which is passed to the action as a parameter. The parameter is optional, so if a third segment is not specified, no default value is passed to the action.

- You can see that when the user requests the URL, **http://www.advworks.com/photo/display/1**, the default route passes the ID parameter 1 to the Display action of the Photo controller.

The following code shows how the default route is implemented in new ASP.NET MVC 5 application

The Default Route

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

Controller Factories and Routes

MVC uses controller factories that help create an instance of a controller class to handle a request. MVC uses action invokers to call the right action and pass parameters to that action method. Both controller factories and action invokers refer to the routing table to complete their tasks.

The following are steps that MVC conducts when a request is received from a web browser:

1. An **MvcHandler** object creates a controller factory. The controller factory is the object that instantiates a controller to respond to the request.

2. The controller factory consults the routing table to determine the right Controller class to use.

3. The controller factory creates a Controller object, and the **MvcHandler** calls the Execute method in that controller.

4. The **ControllerActionInvoker** examines the request URL and consults the routing table to determine the action in the Controller object to call.

5. The **ControllerActionInvoker** uses a model binder to determine the values that should be passed to the action as parameters. The model binder consults the routing table to determine if any segments of the URL should be passed as parameters. The model binder can also pass parameters from a posted form, from the URL query text, or from uploaded files.

6. The **ControllerActionInvoker** runs the action. Often, the action creates a new instance of a model class, perhaps by querying the database with the parameters that the invoker passed to it. This model object is passed to a view, to display results to the user.

As you can see, you can use routes to manage the behavior of controller factories, action invokers, and model binders, because all these objects refer to the routing table. MVC is highly extensible; therefore, developers can create custom implementations of controller factories, action invokers, and model binders. However, by using routes, you can usually implement the URL functionality that you need with the default implementations of these classes. You should ensure that routes cannot implement your required functionality before you plan to customize controller factories, action invokers, or model binders.

Adding and Configuring Routes

Every MVC web application has a **RouteTable** object in which routes are stored, in the Routes properties. You can add routes to the Routes property by calling the MapRoute() method. In the Visual Studio project templates for MVC 5, a dedicated RouteConfig.cs code file exists in the App_Start folder. This file includes the RouteConfig.RegisterRoutes() static method where the default route is added to the RouteTable object. You can add custom routes in this method. The Global.asax.cs file includes a call to RouteConfig.RegisterRoutes() in the Application_Start() method, which means that routes are added to the routing table whenever the MVC application starts.

Properties of a Route

Before you can add a route, you must understand the properties of a route. This is to ensure that these properties match the URL segments correctly and pass requests and parameters to the right location.
The properties of a route include the following:

- **Name**. This string property assigns a name to a route. It is not involved in matching or request forwarding.
- **URL**. This string property is a URL pattern that is compared to a request, to determine if the route should be used. You can use segment variables to match a part of the URL. You can specify a segment variable by using braces.

  For example, if you specify the URL, "photo/{title}", the route matches any request where the relative URL starts with the string, "photo/", and includes one more segment. The segment variable is "title" and can be used elsewhere in the route.
- **Constraints**. Sometimes you must place extra constraints on the route to ensure that it matches only with the appropriate requests. For example, if you want relative URLs in the form, "photo/34", to specify a photo with the ID "34", you must use a URL property like "photo/{id}". However, observe that this URL pattern also matches the relative URL, "photo/create", because it has one extra segment. For IDs, you must constrain the URL to match only segments comprising digits. You can do this by adding a constraint to the route. The Constraints property enables you to specify a regular expression for each segment variable. The route will match a request only if all the segment variables match the regular expressions that you specify.
- **Defaults**. This string property can assign default values to the segment variables in the URL pattern. Default values are used for segment variables when the request does not specify them.

The Default Route Code

The default route specifies the Name, URL, and Defaults properties to obtain controller and action names, and ID values, from the requested relative URL. By examining these properties, you can understand how to construct your own routes.

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

Observe that the URL pattern specifies three segments: {controller}, {action}, and {id}. This means that a relative URL in the form, "photo/display/23", will be sent to the Display action of the PhotoController class. The third segment will be passed as a parameter to the action method. Therefore, if the Display action accepts a parameter named id, the value, 23, will be passed by MVC. In the defaults property, the {id} segment is marked as optional. This means that the route still matches a relative URL, even if no third segment is specified. For example, the route matches the relative URL,"photo/create", and passes the URL to the Create action of the PhotoController class.

The defaults property specifies a default value of "Index" for the {action} segment. This means that the route still matches a relative URL, even if no second segment is specified. For example, the route matches the relative URL, "photo", and passes the URL to the Index action of the PhotoController class.

Finally, the defaults property specifies a default value of "Home" for the {controller} segment. This means that the route still matches a relative URL, even if no segments are specified, that is, if there is no relative URL. For example, the absolute URL, "http://localhost:4565", matches this route, and the URL is passed to the Index action of the HomeController class.

As you can see, with the default route in place, developers build the web application's home page by creating a controller named, HomeController, with an action named, Index. This action usually returns a view called, Index, from the Views/Home folder.

ASP.NET Caching Strategy

Web applications display information on a webpage by retrieving the information from a database. If the information that should be retrieved from the database is large, the application may take longer to display the information on a webpage. ASP.NET MVC 5 supports some caching techniques to help reduce the time required to process a user request.

Before implementing caching, you should first analyze if caching is relevant to your application, because caching is irrelevant to webpages whose content change frequently. To successfully implement caching in your web application, you need to familiarize yourself with the various types of caches, such as output cache, data cache, and HTTP cache.

Why Use Caching?

Caching involves storing the information that is obtained from a database in the memory of a web server. If the content rendered by a webpage is static in nature, the content can be stored in caches or proxy servers. When a user requests content from a web application, caching ensures that the user receives content from the cache, thereby eliminating the need for repeated real time processing.

Caching:

- Reduces the need to repeatedly retrieve the same information from the database.
- Reduces the need to reprocess data, if a user places a request multiple times.
- Helps improve the performance of a web application, by reducing the load on servers.
- Helps increase the number of users who can access the server farm.

However, caching does not help web applications that include frequent content changes. This is because, the content rendered from a cache may be outdated, when compared to the current information. Therefore, you should evaluate the content of your web application and analyze the impact of rendering outdated content, before implementing caching.

The Output Cache

Output cache allows ASP.NET engines to store the rendered content of a webpage in the memory of the web server. Therefore, when a user requests a specific page multiple times, the content is retrieved from the cache, thereby avoiding the execution of programming code in the server. Output cache is a good complement to AJAX partial page updates. Output cache and partial page updates reduce the workload on the server and increase the number of user requests that a server can handle. In ASP.NET MVC 5, you can implement output caching, by adding the OutputCache attribute to the controller.

```
[OutputCache(Duration = 60)]
public PartialViewResult HelloWorld()
{
ViewBag.Message = "Hello World";
return PartialView();
}
```

The OutputCache attribute helps direct the rendering engine to the cache that contains results from the previous rendering process. The Duration parameter of the OutputCache attribute helps control the period of time in seconds for which data should be stored in the cache. By default, the output cache stores only one copy of the rendered content, for each view. Consider a view with the QueryString input parameter that enables content to change based on the variable gathered from the database or a prior request. In this case, you can add the VaryByParam property to the OutputCache attribute, to store a copy of each unique combination of parameters in the cache.

```
OutputCache(Duration = 60, VaryByParam="ID")]
public PartialViewResult HelloWorld()
{
 ViewBag.Message = "Hello World";
 return PartialView();
}
```

In the preceding example, observe that the VaryByParam property refers to QueryString, instead of other MVC parameters. You can also use the VaryByCustom property.

```
[OutputCache(Duration = 60, VaryByCustom="browser")]
public PartialViewResult HelloWorld()
{
 ViewBag.Message = "Hello World";
 return PartialView();
}
```

The Data Cache

Web applications usually depend on the content in a database, to render content on a webpage. Databases sometimes encounter performance issues caused by poorly written queries, which can slow down the performance of database requests resulting in poor webpage performance. You can implement the data cache in your web application to avoid loading data from a database every time a user places a request. The MemoryCache class allows you to implement data cache in your web application. Implementing the data cache involves the following actions:

1. Loading information from the database
2. Storing content in the MemoryCache object
3. Retrieving data from the MemoryCache object
4. Ensuring that content is available in the MemoryCache object; otherwise, reloading the content

The following code shows how to add the MemoryCache object.

```
System.Data.DataTable dtCustomer = System.Runtime.Caching.MemoryCache.Default.
AddOrGetExisting("CustomerData", this.GetCustomerData(),
System.DateTime.Now.AddHours(1));
```

In the preceding example, the following parameters are specified:

- **Key.** The unique identifier of the object that should be stored in the memory cache.
- **Value.** The object that should be stored in the memory cache.
- **AbsoluteExpiration**. The time when the cache should expire.

You can use the AddOrGetExisting function, instead of the Add function, to enable the application to refresh and retrieve data in one line of code. If the cache contains the relevant data, the AddOrGetExisting function retrieves the data from the cache. If the cache does not contain the relevant data, the AddOrGetExisting function allows adding the data to the cache, and then rendering the same data on the webpage.

The HTTP Cache

The Browser Cache

Most web browsers store the content downloaded from web servers in their local cache. Storing data in the local cache helps remove the need to repeatedly download content from the server. Web browsers frequently check content for updates. If the content is updated in the server, web browsers download the content from the server, to attend to user requests. Otherwise, web browsers render content from the local cache.

The Proxy Cache

The functionality of the proxy cache is like the functionality of the browser cache. However, the cache is stored on a centralized server. Users can connect to the Internet or web servers by using this proxy server. Proxy servers store a copy of a web application in a manner like a web browser storing a copy of an application in the local drives. Many users can access the cache in a proxy server, while only one user can access the browser cache at a time. You can implement HTTP caching in the Browser Cache and the Proxy Cache.

Preventing Caching

Caching can sometimes create issues in web applications, because if an application involves frequent content updates, caching prevents users from viewing these content updates. To resolve caching issues, you can implement an HTTP header called **Cache-Control**.

The Cache-Control header indicates to the web browser how to handle the local cache. All HTTP clients, such as browsers and proxy servers, respond to the instructions provided in the Cache-Control header to determine how to handle the local cache of a web application.

You can use the HttpCachePolicy.SetCacheability method to specify the value of the Cache-Control header. The HttpCachePolicy.SetCacheability method helps control caching performance.

The following code shows how to use the HttpCachePolicy.SetCacheability method.

```
Response.Cache.SetCacheability(HttpCacheability.Private);
```

In the preceding example, the HttpCachePolicy.SetCacheability method takes the Private enumeration value.

To prevent caching in your web application, you should set the Cache-Control header value to NoCache. The code shows how to exclude the HTTP cache in your web application.

```
Response.Cache.SetCacheability(HttpCacheability.NoCache);
```

ASP.NET Authentication and Authorization

Authentication is a vital requirement in most web-based applications. Developers usually display only restricted information to all users. Web applications require users to authenticate themselves to view exclusive information. Web applications also display specific information that is relevant to specific user roles. Microsoft ASP.NET includes various authentication models, including local authentication providers, claim-based authentication systems, and federated authentication systems.

Authentication Providers

Authentication providers include code that runs when ASP.NET needs to authorize a user. The code authenticates users by using the information stored in back-end databases, such as Active Directory or Microsoft SQL Server.

You can use ASP.NET Identity with ASP.NET frameworks, such as ASP.NET MVC, Web Forms, Web Pages, Web API, and SignalR. You can use ASP.NET Identity when building web, phone, store, or hybrid applications.

Claims-Based Authentication

Claims-based authentication is a model that facilitates single sign-on. Single sign-on is a feature that allows you to receive a claim when you log on to a centralized authentication system. The claim is a ticket that authentication systems use to authenticate user logons. The claim contains user identity information that helps authentication systems identify users. With claimsbased authentication systems, you can focus efforts on developing business functions, rather than worrying about authenticating users.

Claims-based authentication facilitates:

- Authenticating users to access applications.
- Storing user account information and passwords.
- Checking enterprise directories for user information.
- Integrating the application with the identity systems of other platforms or companies.

To implement claims-based authentication in your web application, you can use Windows Identity Foundation (WIF).

WIF is a set of .NET Framework classes that helps read information from the claims in a web application.

The following steps describe the functioning of claims-based authentication systems:

1. When an unauthenticated user requests a webpage, the request is redirected to the Identity Provider (IP) pages.

2. The IP requires you to enter credentials such as the user name and password.

3. After you enter the credentials, the IP issues a token. Then, the token is returned to the web browser.

4. The web browser is redirected to the webpage that you originally requested.

WIF determines if the token satisfies the requirements to access the webpage. If the token satisfies all requirements, a cookie is issued to establish a session. This cookie ensures that the authentication process occurs only once. Then, the authentication control is passed on to the application. ASP.NET Identity supports claims-based authentication. In ASP.NET Identity a set of claims represents the user's identity.

Federated Authentication

Federations rely on claims-based authentication to allow external parties such as trusted companies to access their applications. Federations use the WS-Federations claim standard to enable the exchange of claims between two parties in a standardized manner.

WIF provides support for federations by using the **WSFederationAuthenticationModule** HTTP module. **WSFederationAuthenticationModule** enables you to implement support for federations in your ASP.NET application, without implementing individual logic. Federated authentication enables Security Token Service (STS) to:

• Process the claims from business partners.

• Extract user information from the claims.

STS enables you to focus more on writing the business logic. It eliminates the need to manage the authentication information of business partners, in your web application. Configuring the WSFederationAuthenticationModule helps specify the STS to which non-authenticated requests should be redirected. WIF provides two methods of federated authentication— FederatedPassiveSignIn and Passive Redirect.

Restricting Access to Resources

You can restrict user access by implementing the Authorize attribute in a controller, instead of using the Web.config file as you would use in an ASP.NET WebForms application. The Web.config file requires physical files to exist, for access control to work. You cannot use the Web.config file to restrict user access because MVC applications route requests to actions, not pages.

The following code shows how to add the Authorize attribute to your controller class.

```
[Authorize()]
public ActionResult GetEmployee()
{ return View(); }
```

Observe the Authorize attribute in the code sample. If you specify this attribute, ASP.NET mandates that users should be authorized to access the view returned by the code sample. If you add the Authorize attribute at the class level, the attribute requires users to log on before they can access anything in the controller class. To allow anonymous users to access a specific portion of your class, you can use the AllowAnonymous attribute.

The following code shows how to use the AllowAnonymous attribute.

```
[AllowAnonymous()]
public ActionResult Register()
{ return View();
}
```

Resilient Web Application

Security is always the top priority for web applications, because publicly accessible web applications are usually the targets of different types of web attacks. Hackers use web attacks to access sensitive information.

Web applications are often subject to security attacks. These attacks prevent the applications from functioning properly, and the attackers try to access sensitive information stored in the underlying data store. ASP.NET provides built-in protection mechanisms that help prevent such attacks. However, such mechanisms also tend to affect the operation of your applications. You should know when to enable or disable the protection mechanisms to avoid any impact on the functionalities of your application. You also need to know how to use Secure Sockets Layer (SSL) to prevent unauthorized access to information during information transmission.

Cross-Site Scripting

Cross-site scripting involves the malicious insertion of scripts in a user session. Cross-site scripting poses information to other websites, by using the authentication information of users, without their knowledge. For example, consider the code inserted into a web application to maliciously post messages on social networking websites, without the knowledge of the users. When a script is inserted into a web application, the script has full access to the Document Object Model (DOM) of the HTML. This access allows the attacker to create fake input boxes in the application, create fake users, and post fake information on the web application.

The cross-site scripting attack usually takes input from improperly escaped output. These scripting attacks usually impact query strings. For example, consider the following URL:

```
http://localhost/Default1/?msg=Hello
```

The following code shows how to access the msg querystring parameter from a controller action.

```
public ActionResult Index(string msg)
{
  ViewBag.Msg = msg;
  return View();
}
```

The following code shows how to display the value of the querystring parameter in a view.

```
<div class="messages">@ViewBag.Msg<div>
```

After running the preceding code samples, the application should display the resultant word, Hello. Now, consider a scenario where the query string parameter msg is changed to a less benign value resulting in the following URL:

```
http://localhost/Default1/?msg=<script>alert('pwnd')</script>
```

As a result, the script block included in the query string is displayed to users. In such cases, attackers can inject malicious code into your app by using the value of a query string parameter. ASP.NET includes request validation, to help protect the input values that are subject to cross-site scripting attacks. However, attackers can bypass this mechanism by using encoding to subvert common cross-site scripting filters. For example, here is the same query string, this time encoded:

```
http://localhost/Default1/?msg=Jon\x3cscript\x3e%20alert(\x27pwnd\x27)%20\x3c/script\x3e
```

You can modify the view class to use the @Ajax.JavaScriptStringEncode function to help prevent crosssite scripting attacks, instead of directly using the input from query strings. The following line of code shows how to use the @Ajax.JavaScriptStringEncode function.

```
<div class="messages">@Ajax.JavaScriptStringEncode(ViewBag.Msg)<div>
```

You can also import the AntiXSS library to check the query string content for possible attacks. The AntiXSS library is part of the Web Protection Library, which was developed by Microsoft to detect more complex web attacks than those that the request validation of ASP.NET can detect. After importing the AntiXSS library in your MVC application, you can use the library to encode any output content in HTML.

The following code shows how to use the AntiXSS library.

```
@using Microsoft.Security.Application
<div class="messages">@Encoder.JavaScriptEncode(ViewBag.Msg)<div>
```

The code in the preceding sample illustrates how to encode input values by using the JavaScriptEncode method of the AntiXSS library, when displaying output in HTML. This practice ensures that the input values are safe for display.

Other Attack Techniques

In addition to cross-site scripting attacks, hackers can use other types of attacks, including cross-site request forgery and SQL injection attacks to subvert web applications.

Cross-Site Request Forgery

Cross-site request forgery (CSRF) is an attack that occurs when you open a URL in a web browser, by using your user context, without knowing that you are allowing attackers to make changes to your system. For example, consider that your application uses query strings to pass information to other applications. You receive an email message with a link such as the following:

```
<a href="http://localhost/Default1/?id=100">Click Me</a>
```

When you click the link, the action associated with the view runs on your web browser. Because, you are an authenticated user in the application, the attacker can now access your system. You can prevent CSRF by using the following rules:

1. Ensure that a GET request does not replay by clicking a link. The HTTP specifications for GET requests imply that GET requests should be used only for retrieval and not for state modifications.

2. Ensure that a request does not replay if an attacker uses JavaScript to simulate a form POST request.

3. Prevent any data modifications that use the GET request. These modifications should require some user interaction. This practice of introducing user interaction does not help prevent form-based attacks. However, user interaction limits several types of easier attacks, such as malicious links embedded in XSS-compromised sites.

The @Html.AntiForgeryToken() function helps protect your system from CSRF by using unique tokens that are passed to the application along with requests. The @Html.AntiForgeryToken() function uses not only a hidden form field but also a cookie value, making it more difficult to forge a request.

The following code shows how to use the @Html.AntiForgeryToken() function in a view.

```
@using (Html.BeginForm())
{ @Html.AntiForgeryToken();
  @Html.EditorForModel();
  <input type="submit" value="Submit" />
}
```

The following code shows how to force Anti-Forgery token checking in a controller by using the ValidateAntiForgeryToken attribute.

```
[HttpPost]
[ValidateAntiForgeryToken()]
public ActionResult Index(User user)
{ Return View(); }
```

Note the ValidateAntiForgeryToken() attribute in the preceding code sample. This attribute enables the controller to check if the user input from the HTML form includes the token generated by the

@Html.AntiForgeryToken(), before accepting a request.

SQL Injection Attack

SQL injection attacks are similar to cross-site scripting attacks. However, the difference is that the user input is used to create dynamic SQL, instead of HTML. Observe the following line of code.

```
cmd.CommandText = "select userID from Users where userID = '" +
Request.QueryString["user"] + "'";
```

Consider a scenario where an attacker modifies the query string as displayed in the following line of code.

```
user=1' AND 1=1 GO SELECT * from Users WHERE NOT 0='
```

The following line of code denotes the resultant SQL.

```
select userID from Users where userID = '1' AND 1=1 GO SELECT * from Users WHERE NOT 0=''
```

In this case, SQL returns everything from the Users table.

To prevent SQL injection attacks, you should:

1. Validate user input.

2. Avoid using string concatenations to create dynamic SQL.

3. Use parameterized commands with dynamic SQL.

4. Store all sensitive and confidential information in encrypted formats.

5. Ensure that the application does not use or access the database with administrator privileges.

Disabling Attack Protection

Request validation is an ASP.NET feature that examines an HTTP request and determines if it contains potentially dangerous content. Potentially dangerous content can include any HTML markup or JavaScript code in the body, header, query string, or cookies of the request. However, request validation can also cause the application to function improperly by preventing some input from entering the application page for processing. Consider a situation in which your application uses an HTML editor to generate HTML code for user input, before saving the input on the database. In this case, you may want to disable the request validation to allow the HTML editor to function properly.

The following code shows how to add the ValidateInput(false) attribute to the controller, to disable request validation for an action in a controller.

```
[HttpPost]
[ValidateInput(false)]
public ActionResult Edit(string comment)
{
return View(comment);
}
```

The following code shows how to add the AllowHtml attribute to the Prop1 property of the model, to disable request validation for this property

```
[AllowHtml]
public string Prop1 { get; set; }
```

You should consider using attack protection techniques that:

• Have a minimum impact on the application.

• Involve minimum fields to accept HTML elements in the request validations.

Secure Sockets Layer

Secure Sockets Layer (SSL) is an application layer protocol that helps:

• Encrypt content by using the public key infrastructure (PKI) keys.

• Protect the content that is transmitted between the server and client.

• Prevent unauthorized access of content during transmission.

• Reassure users that a site is genuine and certified.

You can use SSL on views that accept user input if the input includes sensitive information, such as credit card information and passwords. Using SSL on such crucial views ensures the confidentiality of the content and the authenticity of the sender. However, you may not be able to analyze your code and easily detect if a user accesses the web application by using SSL. ASP.NET MVC 5 includes the RequireHttps attribute that enables you to use SSL on the views that involve sensitive information. The RequireHttps attribute redirects users to the SSL link, if they request a view by using normal HTTP.

The following code shows how to add the RequireHttps attribute in your controller class and action methods to require a user to use the SSL protocol.

```
[RequireHttps]
public class Controller1
{
  [RequireHttps]
  public ActionResult Edit()
  {    }
}
```

You can use the RequireHttps attribute at the controller level or action level. This flexibility allows you to choose SSL when required in your web application. Note that web servers require you to configure the PKI certificate so that the server accepts SSL connections. SSL certificates need to be purchased from a certificate authority (CA). During application development, you can use the self-sign certificate to simplify the configuration process.

Web API

Most web applications require integration with external systems such as mobile applications. You can use the Web API to implement Representational State Transfer (REST) services in your application. REST services help reduce application overhead and limit the data that is transmitted between client and server systems. You need to know how to call Web API services by using server-side code, jQuery code, and JSON.NET library to effectively implement REST-style Web APIs in your application.

Developing a Web API

Web API facilitates creating APIs for mobile applications, desktop applications, web services, web applications, and other applications. By creating a Web API, you make the information in your web application available for other developers to use in their systems. Each web application has a different functional methodology; this difference can cause interoperability issues in applications. REST services have a lightweight design, and Web API helps implement REST services to solve the interoperability issues. You need to know how to use the different routing methods that ASP.NET provides to implement REST services.

What Is a Web API?

Web API is a framework that enables you to build Representational State Transfer (REST)-enabled APIs. REST-enabled APIs help external systems use the business logic implemented in your application to increase the reusability of the application logic. Web API facilitates two-way communication between the client system and the server through tasks such as:

- Instructing an application to perform a specific task
- Reading data values
- Updating data values

Web API enables developers to obtain business information by using REST, without creating complicated XML requests such as Simple Object Access Protocol (SOAP). Web APIs use URLs in requests, thereby eliminating the need for complicated requests. For example, the following URL obtains information for a customer entity with the ID 1: http://localhost:7867/api/customers/1 Web API uses such URLs in requests and obtains results in the JSON format. The following code shows a Web API request response in the JSON format.

A Web API JSON Response

```
[{"Id":1,"Name":"Tomato soup","Category":"Groceries","Price":1.0},
 {"Id":2,"Name":"Yoyo","Category":"Toys","Price":3.75},
 {"Id":3,"Name":"Hammer","Category": "Hardware","Price":16.99}]
```

REST and Web API enable all kinds of different applications, including mobile device applications, to interact with services. Web API provide the following benefits for mobile applications:

- They reduce the processing power needed to create complex request messages for data retrieval.
- They enhance the performance of the application by reducing the amount of data exchange between client and server.

Routing in Web API

When you create a new project by using the Web API template in Visual Studio, it includes a default routing rule. This routing rule helps map HTTP requests to the Web API controllers and actions by using HTTP verbs and the request URL. You can make use of a naming convention to map requests to actions, or you can control the behavior of the mapping by using annotations on action methods.

The Default API Route

Like MVC web applications, Web API applications use routes to map requests to the right API controller and action. In the Visual Studio project templates, the default API route is defined in the WebApiConfig.cs file in the App_Start folder.

```
routes.MapHttpRoute(
name: "API Default",
routeTemplate: "api/{controller}/{id}",
defaults: new { id = RouteParameter.Optional }
);
```

In the preceding code sample, observe that the default route includes the literal path segment api. This segment ensures that Web API requests are clearly separate from MVC controller routes, because Web API requests must start with api. The first placeholder variable, {controller} helps identify the API controller to forward the request to. As for MVC controllers, Web API appends the word, Controller, to this value to locate the right API controller class. For example, Web API routes a request to the URI, api/products, to the controller called, ProductsController. As for MVC controllers, the optional placeholder variable, {id}, is sent to the action as a parameter.

You can also define your own API routes in the same manner as you do for MVC routes. Observe, however, that Web API routes can handle requests from many types of client systems, including mobile device applications, desktop applications, web applications, and web services. MVC routes only handle web browser requests.

Using the Action Naming Convention

The default Web API route does not include a placeholder variable for the action name. This is because Web API uses the HTTP verb and a naming convention to route requests to the right action within a given controller. Clients can make HTTP requests with one of four standard verbs: GET, POST, PUT, and DELETE. Other verbs are possible. Web API looks for an action whose name begins with the requested HTTP verb. For example, if the client sends a DELETE request to the URI api/products/23, Web API looks for a controller called ProductsController. Within this controller, it locates an action whose name begins with Delete. According to the default route, the segment 23 is the {id} parameter. If there is more than one action whose name begins with Delete, Web API chooses the action that accepts a parameter called id. The HttpGet, HttpPut, HttpPost, and HttpDelete Attributes You can use the HttpGet, HttpPut, HttpPost, or HttpDelete attributes in your controller action to override the action naming convention. You can also use these verbs to specify that a function is mapped to a specific HTTP verb.

The HTTPGET Attribute

```
public class ProductsController : ApiController
{
[HttpGet]
public Product FindProduct(id) {}
}
```

Observe that the HTTP attributes only allow mapping of one HTTP verb to an action in the controller.

Visual Studio provides a Web API project template that helps implement Web API in a project. To implement a Web API template in your project, you need to perform the following steps:

1. In the New Project dialog box, click ASP.NET Web Application (.NET Framework).

2. In the New ASP.NET Web Application dialog box, click Web API.

To implement a Web API template in your project, you need to perform the following steps:

After selecting the Web API template, you need to add a new API controller class that derives from ApiController. The API controller class hosts application code for handling Web API requests. ASP.NET engine maps the URL together with the HTTP verb and the controller or the action function of a controller, in the following format.

```
<http verb> http://<hostname>/api/<entity name>/<parameters>
```

The HTTP verb communicates to the Web API about the operations that it should perform; whereas, the rest is to communicate which entity and operations to perform on. Therefore, HTTP plays an important role. For example, consider the following API controller class.

```
public class ProductsController : ApiController
{
public IEnumerable<Product> GetAllProducts() {    }
public Product GetProductById(int id)    {    }
}
```

In the preceding code sample, note the controller classes GetAllProducts and GetProductById. The GetAllProducts controller class helps obtain a full list of products from the database. You can map the

```
GET /api/products
```

The GetProductById controller class helps obtain a specific product by using the ID detail. You can map the following URL with the GetProductById controller class.

```
GET /api/products/id
```

RESTful Services

REST uses URLs and HTTP verbs to uniquely identify the entity that it operates on and the action that it performs. REST helps retrieve business information from the server. However, in addition to data retrieval, business applications perform more tasks such as creating, updating, and deleting information on the database. Web API and REST facilitate handling such additional tasks. They use the HTTP method to identify the operation that the application needs to perform.

HTTP Verb Description

GET Use this method with the following URL to obtain a list of all customers.

```
/api/customers
```

GET Use this method with the following URL to obtain a customer by using the ID detail.

```
/api/customers/id
```

GET Use this method with the following URL to obtain customers by using the category detail.

```
/api/customers?country=country
```

POST Use this method with the following URL to create a customer record.

```
/api/customers
```

PUT Use this method with the following URL to update a customer record.

```
/api/customers/id
```

DELETE Use this method with the following URL to delete a customer record.

```
/api/customers/id
```

Web API allows developers to use a strong typed model for developers to manipulate HTTP request. The following code shows how to use the POST, PUT, and DELETE methods for the create, update, and delete requests to handle the creation, retrieval, updating, and deletion (CRUD) of the customer records.

CRUD Operations

```
public HttpResponseMessage PostCustomer(Customer item) {             }
public void PutCustomer(int id, Customer item) { }
public void DeleteProduct(int id) { }
```

Data Return Formats

When a client makes a request to a Web API controller, the controller action often returns some data. For GET requests, for example, this data might be all the properties of a specific product or all the properties of all the products. Web API can return this data in one of two formats: JavaScript Object Notation (JSON) or XML.

JSON and XML Data Formats

Both JSON and XML are text formats that represent information as strings. You can also use JSON outside the JavaScript code.

```
{"Name":"Albert","Age":29,"Height":145,"Skills":["Programming", "Technical Writing"]}
```

You can represent the same data in the previous code sample, by using XML as shown in the following code.

An XML Response

```
<Employee Name="Albert" Age="29" Height="145">
<Skills>
<Skill Name="Programming" />
<Skill Name="Technical Writing" />
</Skills>
</Employee>
```

When a client makes a request, the client can specify the data format for the response. If the data format is not specified, Web API formats data as JSON by default.

Routes and Controllers in Web APIs

ASP.NET uses a route table to map a URL and an API controller. When you create a project, ASP.NET adds a default route by using the Web API template. This default route helps support the operations of the REST-style Web APIs.

```
routes.MapHttpRoute(
             name: "API Default",routeTemplate:"api/{controller}/{id}",
             defaults: new { id = RouteParameter.Optional });
```

Consider that you want to include multiple actions, such as creating customers with XML and JSON, in the same HTTP method. In this case, you cannot use the default route because it requires a new request for each HTTP method and URL combination. Therefore, you need to update the routing by modifying the Route Map in the WebApiConfig class.

```
routes.MapHttpRoute(
            name: "ActionApi",
            routeTemplate: "api/{controller}/{action}/{id}",
            defaults: new { id = RouteParameter.Optional });
```

You can use the WebApiConfig class to enable multiple versions of API to coexist in the same project. For example, you can include /api/v1/{controller} as Version One of your API and include **`/api/v2/{controller}`** as a new version or Version Two of the API. You may want to include supporting functions in the controller class, and hide the supporting functions from the REST functions. You can eliminate the exposure of the function to the REST interface by adding the NonAction attribute to the action function.

```
[NonAction]
public string GetPrivateData(){ }
```

By default, Web API exposes all public methods as REST services. You can prevent this by making the function private, but this action prevents application code in the same project from accessing the function. Therefore, you can use the NonAction attribute for functions that need to be in public, but do not need to be exposed in REST.

Calling a Web API

After you complete the development of the Web API services, you need to create the client applications to call these services. Calling Web API services is different from calling WCF services. However, the methods that you need to use to call these services are similar, regardless of the platform. You need to know how to call Web API services by using server-side code, jQuery code and JSON.NET library, to effectively implement Web API services in most application platforms.

Calling Web APIs by Using Server-Side Code

You can call REST-style services by using ASP.NET server-side code. You can use the HttpWebRequest class to create a manual HTTP request to the REST services. ASP.NET provides a .NET library that you can use in web applications to call REST-enabled Web API services from the .NET server. To use the .NET library, you need to install the Microsoft.AspNet.WebApi.Client NuGet package. This NuGet package provides

access to the HttpClient class. The HttpClient class simplifies interacting with Web APIs, because it reduces coding efforts. After installing the NuGet package, you need to initialize the HttpClient class. The

```
HttpClient client = new HttpClient();
client.BaseAddress = new Uri("http://localhost/");
client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
```

The last line of code in the preceding code sample informs the client system about the media type that the client system should use. The default media type that applications use is application/json. However, applications can use any other media type, based on the media type that the REST-style services support.

```
HttpResponseMessage response = client.GetAsync("api/customers").Result;

if (response.IsSuccessStatusCode)
{
 var products = response.Content.ReadAsAsync<IEnumerable<Customer>>().Result;
}
else
{    }
```

After running the code in the preceding code sample, you need to define a data model that aligns itself with the one used by the Web API service to enable the .NET library to:

• Process the results of the server-side code.

• Return results as .NET objects for the application to use. Then, you can use the GetAsync and ReadAsAsync methods to create requests to Web API REST services and parse the content into .NET objects. The PostAsJsonAsync function uses the HTTP POST method to call Web API services that support the POST method.

Calling Web APIs by Using jQuery Code

You can call Web API services in the same manner as you call other services that use technologies such as WCF. You can also call Web API services by using the jQuery ajax function.

```
$.ajax({
        url:'http://localhost/api/customers/',
        type: 'GET',
        dataType: 'json',
        success: function (data) { },
        error: function (e) { }
});
```

In the preceding code sample, observe the dataType parameter of the ajax function. You should set this parameter to json or another data type that the Web API service supports. Most applications use JSON because it is light weight. The ajax function has built-in functionalities that parse JSON results for the ease of developers. You can use JSON.stringify() in the data parameter of the ajax function to serialize the JavaScript objects into JSON objects for sending to the Web API method. The following code shows how to use JSON.stringify() in the data parameter of the ajax function.

```
var customer = {
                ID:'1',
                CustName: 'customer 1'
};

$.ajax({url: 'http://localhost/api/customer',
        type: 'POST',
        data:JSON.stringify(customer),
        contentType: "application/json;charset=utf-8",
        success: function (data) {  },
        error: function (x) {
       }}
);
```

ASP.NET Filters

ASP.NET MVC provides a simple way to inject your piece of code or logic either before or after an action is executed. This is achieved by decorating the controllers or actions with ASP.NET MVC attributes or custom attributes. An attribute or custom attribute implements the ASP.NET MVC filters (filter interface) and can contain your piece of code or logic. You can make your own custom filters or attributes either by implementing ASP.NET MVC filter interface or by inheriting and overriding methods of ASP.NET MVC filter attribute class if available.

Typically, Filters are used to perform the following common functionalities in your ASP.NET MVC application.

1. **Custom Authentication**
2. **Custom Authorization (User based or Role based)**
3. **Error handling or logging**
4. **User Activity Logging**
5. **Data Caching**
6. **Data Compression**

The ASP.NET MVC framework provides five types of filters.

1. **Authentication filters**
2. **Authorization filters**
3. **Action filters**
4. **Result filters**
5. **Exception filters**

Authentication Filters

This filter is introduced with ASP.NET MVC5. The IAuthenticationFilter interface is used to create CustomAuthentication filter. The definition of this interface is given below:

```
public interface IAuthenticationFilter
{
 void OnAuthentication(AuthenticationContext filterContext);

 void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext);
}
```

You can create your CustomAuthentication filter attribute by implementing IAuthenticationFilter as shown below-

```
public class CustomAuthenticationAttribute : ActionFilterAttribute, IAuthenticationFilter
{
 public void OnAuthentication(AuthenticationContext filterContext)
 {
 //Logic for authenticating a user
 }
 //Runs after the OnAuthentication method
 public void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext)
 {
 //TODO: Additional tasks on the request
 }
}
```

Authorization Filters

The ASP.NET MVC Authorize filter attribute implements the IAuthorizationFilter interface. The definition of this interface is given below-

```
public interface IAuthorizationFilter
{
 void OnAuthorization(AuthorizationContext filterContext);
}
```

The AuthorizeAttribute class provides the following methods to override in the CustomAuthorize attribute class.

```
public class AuthorizeAttribute : FilterAttribute, IAuthorizationFilter
{
 protected virtual bool AuthorizeCore(HttpContextBase httpContext);
 protected virtual void HandleUnauthorizedRequest(AuthorizationContext filterContext);
 public virtual void OnAuthorization(AuthorizationContext filterContext);
 protected virtual HttpValidationStatus OnCacheAuthorization(HttpContextBase
httpContext);
}
```

In this way you can make your CustomAuthorize filter attribute either by implementing IAuthorizationFilter interface or by inheriting and overriding above methods of AuthorizeAttribute class.

Action Filters

Action filters are executed before or after an action is executed. The **IActionFilter** interface is used to create an Action Filter which provides two methods **OnActionExecuting** and **OnActionExecuted** which will be executed before or after an action is executed respectively.

```
public interface IActionFilter
{
 void OnActionExecuting(ActionExecutingContext filterContext);
 void OnActionExecuted(ActionExecutedContext filterContext);
}
```

Result Filters

Result filters are executed before or after generating the result for an action. The Action Result type can be ViewResult, PartialViewResult, RedirectToRouteResult, RedirectResult, ContentResult, JsonResult, FileResult and EmptyResult which derives from the ActionResult class. Result filters are called after the Action filters. The IResultFilter interface is used to create an Result Filter which provides two methods OnResultExecuting and OnResultExecuted which will be executed before or after generating the result for an action respectively.

```
public interface IResultFilter
{
 void OnResultExecuted(ResultExecutedContext filterContext);
 void OnResultExecuting(ResultExecutingContext filterContext);
}
```

Exception Filters

Exception filters are executed when exception occurs during the actions execution or filters execution. The IExceptionFilter interface is used to create an Exception Filter which provides OnException method which will be executed when exception occurs during the actions execution or filters execution.

```
public interface IExceptionFilter
{
 void OnException(ExceptionContext filterContext);
}
```

ASP.NET MVC HandleErrorAttribute filter is an Exception filter which implements IExceptionFilter. When HandleErrorAttribute filter receives the exception it returns an Error view located in the Views/Shared folder of your ASP.NET MVC application.

Order of Filter Execution

All ASP.NET MVC filter are executed in an order. The correct order of execution is given below:

1. **Authentication filters**
2. **Authorization filters**
3. **Action filters**
4. **Result filters**

Configuring Filters

You can configure your own custom filter into your application at following three levels:
Global level
By registering your filter into Application_Start event of Global.asax.cs file with the help of FilterConfig class.

```
protected void Application_Start()
{  FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters); }
```

Controller level
By putting your filter on the top of the controller name as shown below-

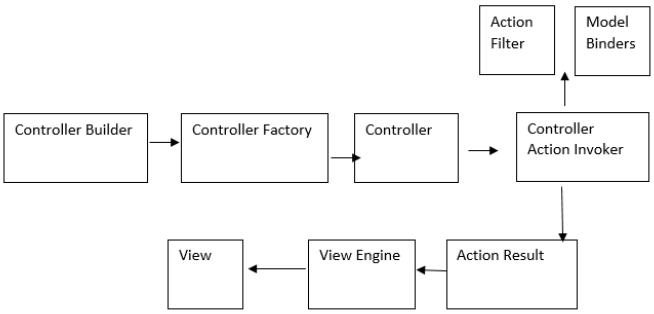
```
[Authorize(Roles="Admin")]
public class AdminController : Controller
{   //   }
```

Action level
By putting your filter on the top of the action name as shown below-

```
public class UserController : Controller
{ [Authorize(Users="User1,User2")]
  public ActionResult LinkLogin(string provider)
  {  // TODO:
      return View();
  }
}
```

ASP.NET MVC Extensibility

Any of these ASP.NET MVC components can be replaced.



Dependency Injection Improvements

ASP.NET MVC provides better support for applying Dependency Injection (DI) and for integrating with Dependency Injection or Inversion of Control (IOC) containers.

Support for DI has been added in the following areas:

- Controllers (registering and injecting controller factories, injecting controllers).
- Views (registering and injecting view engines, injecting dependencies into view pages).
- Action filters (locating and injecting filters).
- Model binders (registering and injecting).
- Model validation providers (registering and injecting).
- Model metadata providers (registering and injecting).
- Value providers (registering and injecting).

Model Binding

Model binding is the process of creating .NET objects using the data sent by the browser in an HTTP request.
Default model binder search in following location and order for named parameters data.

Name	Description
Request.Form	Content HTML form element data
RouteData.Values	Application routes values
Request.QueryString	Data in the query string of the request URL
Request.Files	Files that have been uploaded as part of the request

Bind attribute:

- To include or exclude model properties from the binding process.
- To include first and last name in person object. Person FirstName and LastName property value will be consider other will ignored.

```
public ActionResult Register([Bind(Include = "FirstName, LastName")]Person  person)
```

Example: To exclude sex property in person object.
Person sex property value will be ignored.

```
public ActionResult Register([Bind(Exclude = "Sex")] Person person)
```