Programming ASP.NET Core

Chapter 01: The First ASP.NET Core Project

1. Write main features of .Net core platform.

Answer:

2. What is OWIN?

Answer:

3. What is .NET CLI?

Answer: In .NET Core, the entire set of fundamental development tools-those used to build, test, run, and publish applications-is also available as command-line applications. Together, such applications are referred to as the .NET Core Command-line Interface (CLI).

4. What is .NET driver?

Answer: The CLI is generally referred to as a collection of tools, but instead, it is a collection of commands run by a host tool known as the driver. This tool is dotnet.exe (see Figure 1- Any command-line instruction takes the following form:
dotnet [host-options] [command] [arguments] [common-options]
The [command] placeholder refers to the command to execute within the driver tool whereas [arguments] refers to the arguments being passed to the command.

5. What is the use of the global.json file?

Answer: A route is a URL template that your application can recognize and process. A route is ultimately mapped to a pair of controller and action names. As we'll see in a moment, you can add as many routes as you wish, and those routes can take nearly any shape you like them to be. An internal MVC service is responsible for request routing; it is automatically registered when you enable MVC Core services.

6. Write down three .NET CLI Command?

Answer:

a) dotnet new,
b) dotnet restore,
c) dotnet build

Chapter 02: The First ASP.NET Core Project

1. **Write down the purpose of the wwwroot folder in ASP.NET Core web application?**

Answer: As far as static files are concerned, the ASP.NET Core runtime distinguishes between content root folder and the web root folder.
The content root is generally the current directory of the project, and in production, it is the root folder of deployment. It represents the base path for any file search and access that might be required by the code. Instead, the web root is the base path for any static files that the application might serve to web clients. Generally, the web root folder is a child folder of the content root and is named wwwroot.
The interesting thing is that the web root folder must be created on the production machine, but it is completely transparent to client browsers requesting static files. In other words, if you have an images subfolder below wwwroot with a file named banner .jpg, then the valid URL to grab the banner is the following:
However, the physical image file must go under wwwroot on the server; otherwise, it won't be retrieved. The location of both root folders may be changed programmatically in the program.cs file. (More on this in a moment.)

2. **What is the purpose of the Startup.cs file in ASP.NET Core web application?**

Answer:
The startup.cs file contains the class designated to configure the request pipeline that handles all requests made to the application. The class contains at least a couple of method that the host will call back during the initialization of the application.

3. **Write down the use of ConfigureServices and Configure method in the Startup.cs file?**

Answer: Configure Services is used to add in the dependency injection mechanism services that the application expects to use. The ConfigureServices is optional to have in a startup class, but having one is necessary in most realistic scenarios.
The second method is called Configure and, as its name suggests, serves the purpose of configuring previously requested services. For example, if you declared your intention to us the ASP.NET MVC service in the method ConfigureServices, then in Configure you can specify the list of valid routes you intend to handle by calling the UseMvc method on the provided IApplicationBuilder parameter. The Configure method is required. Note that the startup class is not expected to implement any interface or inherit from any base class. Both Configure and ConfigureServices, in fact, are discovered and invoked via reflection.

4. **What is Dependency Injection?**

Answer: To use the DI system, you need to register the types the system must be able to instantiate for you. The ASP.NET Core DI system is already aware of some types, such as IHosting Environment and ILoggerFactory, but it needs to know about application-specific types.

Chapter 03: Bootstrapping ASP.NET MVC

1. What is a route?

Answer:

2. What are Route Constraints?

Answer:

Predefined route constraints

Mapping Name	Class	Description
Int	IntRouteConstraint	Ensures the route parameter is set to an integer
Bool	BoolRouteConstraint	Ensures the route parameter is set to a Boolean value
datetime	DateTimeRouteConstraint	Ensures the route parameter is set to a valid date
decimal	DecimalRouteConstraint	Ensures the route parameter is set to a decimal
double	DoubleRouteConstraint	Ensures the route parameter is set to a double
Float	FloatRouteConstraint	Ensures the route parameter is set to a float
Guid	GuidRouteConstraint	Ensures the route parameter is set to a GUID
Long	LongRouteConstraint	Ensures the route parameter is set to a long integer
minlength(N)	MinLengthRouteConstraint	Ensures the route parameter is set to a string no shorter than the specified length.
maxlength(N)	MaxLengthRouteConstraint	Ensures the route parameter is set to a string no longer than the specified length
length(N)	LengthRouteConstraint	Ensures the route parameter is set to a string of the specified length
min(N)	MinLengthRouteConstraint	Ensures the route parameter is set to an integer greater than the specified value
max(N)	MaxLengthRouteConstraint	Ensures the route parameter is set to an integer smaller than the specified value
range(M, N)	RangeRouteConstraint	Ensures the route parameter is set to an integer that falls within the specified range of values
alpha	AlphaRouteConstraint	Ensures the route parameter is set to a string made of alphabetic characters
regex(RE)	RegexInlineRouteConstraint	Ensures the route parameter is set to a string compliant with the specified regular expression.
required	RequiredRouteConstraint	Ensures the route parameter has an assigned value in the URL.

3. What are Action Filters?

Answer: An action filter is a piece of code that runs around the execution of a controller method. The most common types of action filters are filters that run before or after the controller method executes. For example, you can have an action filter that only adds an HTTP header to a request or an action filter that refuses to run the controller method if the request is not coming via Ajax or from an unknown IP address or referrer URL.

Chapter 04: ASP.NET MVC Controllers

1. What are actions in the controller?

Answer:

CONTROLLER ACTIONS

The final output of the route analysis of the URL of an incoming request is a pair made of the name of the controller class to instantiate and the name of the action to perform on it. Executing an action on a controller invokes a public method on the controller class. Let's see how action names are mapped to class methods.

Mapping Actions to Methods

The general rule is that any public method on a controller class is a public action with the same name. As an example, consider the case of a URL like /home/index. Based on the routing facts we have discussed earlier, the controller's name is "home," and it requires an actual class named Home Controller available in the project. The action name extracted from the URL is "index." Subsequently, the Home Controller class is expected to expose a public method named Index.

There are some additional parameters that might come into play, but this is the core rule of mapping actions to methods.

Mapping by Name

To see all aspects of action-to-method mapping in the MVC application model, let's consider the following example.

```
public class Home Controller: Controller
{ // Implicit action name: Index public ActionResult Index()
    {
            ....
    }
    [NonAction]
    public ActionResult About()
    {
            ...
    } [ActionName("About")]
    public ActionResult LoveGerman Shepherds ()
    {
            ...
    }
}
```

2. How do you map an action to HTTP Verbs?

Answer: The MVC application model is flexible enough to let you bind a method to an action only for a specific HTTP verb. To associate a controller method with an HTTP verb, you the parametric Accept Verbs attribute or direct attributes such as HttpGet, HttpPost, and HttpPut.

3. What is attribute based routing?

Answer: Attribute-based routing is an alternate way of binding controller methods to URLS. The idea is that instead of defining an explicit route table at the startup of the application, you decorate controller methods with ad hoc route attributes. Internally, the route attributes will populate the system's route table.

4. What is model binding?

Answer: Using native request collections of input data works but from a readability and maintenance standpoint, it is preferable to use an ad hoc model to expose data to controllers. This model is sometimes referred to as the input model. ASP.NET MVC provides an automatic binding layer that uses a built-in set of rules for mapping raw request data from a variety of value providers to properties of input model classes. As a developer, you are largely responsible for the design of input model classes.

5. What are Action filters?

Answer: An action filter is a piece of code that runs around the execution of an action method can be used to modify and extend the behavior coded in the method itself.

An action filter is fully represented by the following interface:

```
public interface IActionFilter
{

    void OnActionExecuting (Action ExecutingContext filterContext);
    void OnActionExecuted(ActionExecutedContext filterContext);
}
```

Chapter 05: ASP.NET MVC Views

1. What are areas in MVC applications?

Answer: Areas are a feature of the MVC application model used to group related functionalities within the context of a single application. Using areas is comparable to using multiple sub- applications, and it is a way to partition a large application into smaller segments.

The partition that areas offer is analogous to namespaces, and in an MVC project, adding an area (which you can do from the Visual Studio menu) results in adding a project folder where you have a distinct list of controllers, model types, and views. This allows you to have two or more HomeController classes for different areas of the application. Area partitioning is up to you and is not necessarily functional. You can also consider using areas one-to-one with roles.

In the end, areas are nothing technical or functional; instead, they're mostly related to the design and organization of the project and the code. When used, areas have an impact on routing. The name of the area is another parameter to be considered in the conventional routing. For more information refer to http://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/areas.

2. What is the use of the _ViewImports.cshtml?

Answer: a Razor view results from the combination of multiple.cshtml files, such as the view itself, the layout file, and two optional global files named_viewstart.cshtml and_viewimports.cshtml. The role of these two files is explained below.

File name	Purpose
ViewStart.cshtml	Contains code that is being run before any view is rendered. You can use this file to add any configuration code that is common to all views in the application. You commonly use this file to specify a default layout file for all views This file must be located in the root Views folder and is also supported in classic ASP.NET MVC
_ViewImports.cshtml	Contains Razor directives that you want to share across all views. You're allowed to have multiple copies of this file in various view folders. The scope of its content affects all views in the same folder or below it unless another copy of the file exists at an inner level. This file is not supported in classic ASP.NET. In classic ASP.NET, though, the same purpose is achieved using a web.config file

3. What are ViewData and ViewBag?

Answer: ViewData is a classic name/value dictionary. The actual type of the property is View DataDictionary which is not derived from any of the system's dictionary types but still exposes the common dictionary interfaces as defined in the .NET Core framework.

The base Controller class exposes a ViewData property, and the content of that property is automatically flushed into the dynamically created instance of the Razor Page<T> class behind the view. This means that any value stored in controller ViewData is available in the view without any further effort on your end.

```
public IActionResult Index()
    {
            ViewData["PageTitle"] = "Hello";
            ViewData["Copyright"]"(c) Dino Esposito";
            ViewData["CopyrightYear"] - 2017;
            return View();
    }
```

Chapter 06: The Razor Syntax

1. What are tag helpers?

Answer: An HTML helper is an extension method of the HtmlHelper class. Abstractly speaking, an HTML helper is nothing more than an HTML factory. You call the method in your view; some HTML is inserted that results from provided input parameters (if any). Internally, an HTML helper simply accumulates markup into an internal buffer and then outputs it. A view object incorporates an instance of the HtmlHelper class under the property name, Html.

ASP.NET Core supplies a few HTML helpers out of the box, including CheckBox, ActionLink, and TextBox.

Method	Type	Description
BeginForm, BeginRoute Form	Form	Returns an internal object that represents an HTML form that the system uses to render the <form> tag
EndForm	Form	A void method, closes the pending </form> tag
CheckBox, CheckBoxFor	Input	Returns the HTML string for a check box input element
Hidden, Hidden For	Input	Returns the HTML string for a hidden input element
Password, PasswordFor	Input	Returns the HTML string for a password input element
RadioButton, RadioButtonFor	Input	Returns the HTML string for a radio button input element
TextBox, TextBoxFor	Input	Returns the HTML string for a text input element
Label, LabelFor	Label	Returns the HTML string for an HTML label element
ActionLink, RouteLink	Link	Returns the HTML string for an HTML link
DropDownList	List	Returns the HTML string for a drop-down list
DropDownListFor ListBox, ListBoxFor	List	Returns the HTML string for a list box
TextArea, TextArea For	TextArea	Returns the HTML string for a text area

2. What are sections in layout?

Answer:
Any layout is forced to have at least one injection point for external view content. This injection point consists of a call to the method Render Body. The method is defined in the base view class being used to render layouts and views. Sometimes, though, you need to inject content into more than one location. In this case, you define one or more named sections in the layout template and let views fill them out with markup.

```
<body>
    <div class="page">
            @RenderBody ()
    </div>
    <div id="footer">
            @Render Section("footer")
    </div>
</body>
```

Each section is identified by name and is considered required unless it is marked as optional. The Render Section method accepts an optional Boolean argument that denotes whether the section is required. To declare a section optional, you do as follow:

```
<div id="footer">
    @Render Section ("footer", false)
</div>
```

The following code is functionally equivalent to the preceding code, but it's much better from a readability standpoint:

```
<div id="footer">
    @Render Section ("footer", required: false)
</div>
```

3. **What are partial views?**
Answer: A partial view is a distinct piece of HTML that is contained in a view, but it is treated as an entirely independent entity. In fact, it is even legitimate to have a view written for one view engine and a referenced partial view that requires another view engine. Partial views are like HTML subroutines and serve two main scenarios: Having reusable UI-only HTML snippets and breaking up complex views into smaller and more manageable pieces.

4. **What are view components?**
Answer: View components are a relatively new entry in the world of ASP.NET MVC. Technically, they are self-contained components that include both logic and view. In this regard, they're a revised version, and a replacement, of child actions as they appeared in classic ASP.NET.

Chapter 07: Design Considerations

1. Write down the Lifetime options for DI-created instances?

Answer: In ASP.NET Core, there are a few different ways to request the DI system an instance of the mapped concrete type.

Method	Description
Add Transient	The caller receives a new instance of the specified type per call
AddSingleton	The caller receives the same instance of the specified type which was created the first time. Regardless of the type, every application gets its own instance
AddScoped	Same as AddSingleton, except that it is scoped to the current request

Note that by simply using an alternate overload of the AddSingleton method you can also indicate the specific instance to be returned for any successive calls. This approach is helpful when you need the object being returned to be configured with a certain state.

Chapter 08: Securing the Application

1. How do you enable authentication in MVC Core web applications?

Answer: Enabling Authentication Middleware To enable cookie authentication in a brand new ASP.NET Core application, you need to reference the Microsoft. AspNetCore. Authentication. Cookies package. The actual code entered into the application, however, is different in ASP.NET Core 2.0 compared to what Was in earlier versions of the same ASP.NET Core framework.

2. How do you enable authentication on a controller?

Answer:
The Authorize attribute is the declarative way to secure a controller or just some of its methods.

```
[Authorize]
public class Customer Controller : Controller
    {
            …
    }
```

Note that if specified without arguments, the Authorize attribute only checks if the user is authenticated. In the code snippet above, all users who can successfully sign in to the system are equally enabled to call into any methods of the Customer Controller class. To select only a subset of users, you use roles.

The Roles property on the Authorize attribute indicates that only users in any of the listed roles would be granted access to the controller methods. In the code below, both Admin and System users are equally enabled to call into the Backoffice Controller class.

```
[Authorize (Roles="Admin, System")]

public class BackofficeController: Controller

{
…
    [Authorize (Roles="System")]

    public IAction Result Reset()

    {
    // You MUST be a SYSTEM user to get here
    …
    }
    [Authorize]
    public ActionResult Public()
    {
    // You just need be authenticated and can view this
    // regardless of role(s) assigned to you
    …
    }
    [AllowAnonymous)]

    public IActionResult Index()
    {
    // You don't need to be authenticated to get here
    ...
    }
}
```

The Index method doesn't require authentication at all. The Public method just requires an authenticated user. The method Reset strictly requires a System user. All other methods you might have work with either an Admin or a System user.

If multiple roles are required to access a controller, you can apply the Authorize attribute multiple times. Alternatively, you can always write your own authorization filter. In the code below, only users who have the Admin and the System role will be granted permission to call into the controller.

```
[Authorize (Roles="Admin")]
[Authorize (Roles="System")]
public class BackofficeController Controller
{
        …
}
```

Optionally, the Authorize attribute also can accept one or more authentication schemes through the Active Authentication Schemes property.

```
[Authorize (Roles="Admin, System", ActiveAuthentication Schemes="Cookies"]
public class BackofficeController Controller
{
```

Chapter 09: Access to Application Data

1. **How do you Configure DbContext of EF core dependency injection?**

Answer:

Retrieving the Connection String

The way in which EF6 context classes retrieve their connection string is not completely compatible with the newest and totally rewritten configuration layer of ASP.NET Core. Let's consider the following common code fragment.

```
public class MyOwnDatabase: DbContext
{
    public MyOwnDatabase (string connStringOr DbName = "name=MyOwnDatabase")
    : base(connStringOr DbName)
}
```

The application-specific Db context class receives the connection string as an argument or retrieves it from the web.config file. In ASP.NET Core, there's nothing like a web.config file, so the connection string either becomes a constant or should be read through the .NET Core configuration layer and passed in.

Integrating EF Context with ASP.NET Core DI

Most of the ASP.NET Core data access examples you find on the web show how to inject the DB context into all layers of the application via Dependency Injection (DI). You can inject the EF6 context in the DI system as you would do with any other service. The ideal scope is per-request, which means the same instance is shared by all possible callers within the same HTTP request.

At the same time, you should be aware that deploying several self-contained applications to a system can absorb large amounts of disk space because the entire .NET Core Framework is duplicated on a per-application basis.

To support the self-contained deployment of a given application, you have to explicitly add the runtime identifiers of the platforms you intend to support. When Visual Studio creates a new .NET Core project, this information is missing and results in a portable deployment. To enable self-contained deployment, you have to manually edit the .csproj project file and add a RuntimeIdentifiers node to it.

Here's the .csproj content for the sample project.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
    <PropertyGroup>
            <TargetFramework>netcoreapp2.0</TargetFramework>
<RuntimeIdentifiers>win10-x64; linux-x64</RuntimeIdentifiers>
    </PropertyGroup>
    <ItemGroup>
            <None Remove="Properties\Publish Profiles\Folder Profile.pubxml" />
</ItemGroup>
<ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
</ItemGroup>
<ItemGroup>
    <DotNetCliTool Reference
            Include="Microsoft.VisualStudio. Web.CodeGeneration. Tools" Version="2.0.0" />
</ItemGroup>
<ItemGroup>
    <Folder Include="Pages\Shared\" />
    <Folder Include="Properties\PublishProfiles\" />
</ItemGroup>
</Project>
```

Chapter 10: Designing Web API

1. What is a Web API?

Answer: In the context of AS P.NET Core, the term
-web API" finally gets its real meaning without ambiguity and need to explain the contours further. A web API is a programmatic interface made of a number of publicly exposed HTTP endpoints that typically (but not necessarily) return JSON or XML data to callers. A web API fits nicely in what today appears to be a fairly common application scenario: a client application needs to invoke some remote back end to download data or request processing.
The client application can take many forms including a JavaScript-intensive web page. A rich client, or a mobile application. In this chapter, we'll see what it takes to build a web API in ASP.NET Core. In particular, we'll focus on the philosophy of the API—whether REST oriented or procedure-oriented—and on how to secure it.

2. How do you return JSON data to client from a controller?

Answer: To return JSON data, all you do is create an ad hoc method in a new or existing Controller class. The sole specific requirement for the new method is returning a JsonResult object.

```
Public IActionResult LatestNews(int count)
{
    var listOfNews = Service. GetRecentNews(count);
    return Json(listOfNews);
}
```

The Json method ensures that the given object is packaged in a JsonResult object. Once returned from the controller class, the JsonResult object is processed by the action invoker, which is when the actual serialization takes place. That's all of it. You retrieve the data you need; you package it up into an object, and pass it on to the Json method. Done, Or, at least, it's done if the data is fully serializable. The actual URL to invoke the endpoint can be determined through the usual routing approaches—conventional routing and/or attribute routing.

3. What is a REST service?

Answer: REST is a very common approach to unify the way public APIs are exposed to clients
ASP.NET Core controllers just support some extra features to make the output as RESTful
as possible.

REST at a Glance

The core idea behind REST is that the web application—mostly a web API—works entirely based on the full set of capabilities of the HTTP protocol, including verbs, headers, and status codes. REST is a shorthand name for Representational State Transfer, which means the application will handle requests in the form of HTTP verbs (GET, POST, PUT, DELETE, and HEAD) acting on resources. In REST, a resource is nearly identical to a domain entity and is represented by a unique URI.

4. Write down the http verb commonly used in REST service?

Answer:

Method	Description
DELETE	Issues the request for deleting the addressed resources, whatever that means for the back end. The Actual implementation of the "delete" operation belongs to the application and could be either something physical or logical.
GET	Issues the request for getting the current representation of the addressed resource. The use of additional HTTP headers can fine-tune the actual behavior. For example, the If-Modified-Since header mitigates the request by expecting a response only if changes have occurred since the specified time.
POST	Issues the request for adding a resource when the URT is not known in advance. The REST response to this request returns the URI of the newly created resource. Again,

	what “adding a resource” actually means for the back end is the responsibility of the back end.
PUT	Issues the request for making sure that the state of the addressed resource is in line with provided information. It’s the logical counterpart of an update command.

5. What is token-based authentication?

Answer:

Token-based Authentication

The idea is that the web API receives an access token—typically a GUID or an alphanumeric string—validates it and serves the request if the token is not expired and is valid for the application. There are various ways to issue a token. The simplest is that tokens are issued offline when a customer contacts the company to license the API. You create the token and associate it with a particular customer. From that point forward, the customer is responsible for the abuse or misuse of the API, and server-side methods work only if they recognize the token.

The web API back end needs to have a layer that checks tokens. You can add this layer as plain code to any method or, better yet, configure it to be a piece of the application middleware. Tokens can be appended to the URL (for example, as query string parameters) or embedded in the request as an HTTP header. None of these approaches is perfect, and no approach exists that is safer. In both cases, the value of the token can be spied on. Using a header is preferable because an HTTP header is not immediately visible in the URL.

To make the defense stronger, you might want to use some strict expiration policy on the tokens. All in all, though, the strength of this approach is that you always know who is responsible for the abuse or misuse of the API and can stop them from disabling the token at any time.

Chapter 11: Posting Data from the Client Side

1. How do you upload file to the server in ASP.NET Core?

Answer:

Uploading Files to a Web Server

In HTML, files are pretty much treated like any other type of input in spite of the deep difference that exists between files and primitive data. As usual, you start by creating one or more INPUT elements with the type attribute set to file. The native browser user interface allows users to pick a local file, and then the content of the file is streamlined with the rest of the form content. On the server, the file content is mapped to a new type the IformFile type—and enjoys a much more uniform treatment from the model binding layer than in previous versions of MVC.

Chapter 12: Client-side Data Binding

1. **How do you define refreshable are in a page?**

Answer: There is no doubt that a full refresh of a webpage that is quite rich with graphics and media can be significantly slow and cumbersome for users. This is precisely why Ajax and partial rendering of pages became so popular. At the other extreme of page rendering, we find the concept of a Single Page Application (SPA). At its core, an SPA is an application made of one (Or a few) minimal HTML pages incorporating a nearly empty DIV populated at runtime with a template and data downloaded from some server. On the way from server-side rendering to the full client-side rendering of SPAs, I suggest we start with an HTML partial rendering approach.

Chapter 13: Building Device-friendly Views

1. What does the Modernizer do?

Answer: The idea behind feature detection is simple and, to some extent, even smart. You don't even attempt to detect the actual capabilities of the requesting device, which is known to be cumbersome, difficult, and even poses serious issues as far as maintainability of the solution is concerned. Modernizer consists of a JavaScript library with some code that runs when page loads and checks whether the current browser can offer certain HTML5 and CSS3 functionalities. Modernizer exposes its findings programmatically so that the code in the page can query the library and intelligently adapt the output.

2. Why do you use WURFL.JS.

Answer: In spite of the name, WURFL.JS is not a static JavaScript file you can host On-premises or upload to your cloud site. More precisely, WURFL.JS is an HTTP endpoint you link to your
web views through a regular SCRIPT element.

To get the WURFL.JS services, therefore, you only need to add the following line to an HTML view you have that need to know about the actual device.

```
script type="text/javascript" src="//wurfl.io/wurfl.js"></script>
```

The browser knows nothing about the nature of the WURFL.JS endpoint. The browser Just attempts to download and execute any script code it can get from the specified URL the WURFS server that receives a request uses the user agent of the calling device to figure out its actual capabilities. The WURFL server relies on the services of the WURFL framework—a powerful device data repository and a cross-platform API used by Facebook, Google, and PayPal).

Chapter 14: The ASP.NET Core Runtime Environment

1. **What does the WebHost class do?**

Answer: WebHost is a static class that provides two methods for creating instance of classes exposing the IwebHostBuilder interface with predefined settings. The class also comes with many methods to quickly start the environment passing just the URL to listen to and a delegate for the behavior to implement. Again, this is the living proof of the extreme flexibility of the ASP.NET Core runtime.

2. **What is a Reverse Proxy?**

Answer: Configuring a Reverse Proxy
Originally, the Kestrel server was not designed to be exposed to the open Internet, Meaning that a reverse proxy was required on top of it for security reasons and as a way to protect the application from possible web attacks. Starting with ASP.NET Core 2.0, though, a thicker defense barrier was added, resulting in more configuration options to take into account.

Chapter 15: Deploying an ASP.NET Core Application

1. What is an ASP.NET Core Application?

Answer: Writing an ASP.NET Core application requires the creation and editing of a variety of files, not all of which are really necessary to put the application live on a production or staging server. Hence, the very first step on the way to deploying an ASP.NET Core application is publishing it to a local folder so that all necessary files are compiled, and only those files that need be moved to the live environment are isolated somewhere. The list of deployable files usually includes code files compiled to DLLs plus static and configuration files.

Classic ASP.NET applications could only be deployed to IIS under a Windows server operating system and more recently to a Microsoft Azure app service. For ASP.NET Core applications, you have more choices, including a Linux on-premise machine or another cloud environment such as Amazon Web Services (AWS) or even a Docker container.

2. What containers in an ASP.NET Core Application deployment?

Answer: Publishing Self-contained Applications Publishing a portable application has been the norm for the entire lifetime of the ASP.NET platform. The size of deployment is small and limited to the sole application binaries and files. On the server, multiple applications share the same framework binaries. With .NET Core, the alternative to a portable deployment is to publish self-contained applications. When a self-contained application is published, the .NET Core binaries for the specified runtime environment are also copied over. This makes the size of the deployment significantly larger. For the sample application discussed here, the size of a portable deployment is less than 2 MB, but it can grow up to 90 MB for a self-contained install that targets a generic Linux platform. The upside of self-contained applications, however, is that the application has everything it needs to run regardless of the version(s) of the .NET Core Framework installed on the