

Unit 4

Creating ASP.NET Core MVC Applications

Creating a Web App & Run

- From the Visual Studio, select Create a new project.
- Select ASP.NET Core Web Application and then select Next.
- Name the project as you like or WebApplicationCoreS1
- Choose the location path to save your project.
- Click Create
- Select Web Application(Model-View-Controller), and then select Create.
- Now, To run the App,
 - Select **Ctrl-F5** to run the app in non-debug mode, Or
 - Select IIS Express Button.



Setting up the Environment

ASP.NET Core wwwroot Folder

- By default, the wwwroot folder in the ASP.NET Core project is treated as a web root folder. Static files can be stored in any folder under the web root and accessed with a relative path to that root.
- In ASP.NET Core, only those files that are in the web root - wwwroot folder can be served over an http request. All other files are blocked and cannot be served by default.
- Generally, we find static files such as JavaScript, CSS, Images, library scripts etc. in the wwwroot folder
- You can access static files with base URL and file name. For example, for css folder, we can access via `http://localhost:<port>/css/app.css`

Setting up the Environment

ASP.NET Core – Program.cs Class

- ASP.NET Core web application project starts executing from the entry point - public static void Main() in Program class.

ASP.NET Core – Startup.cs Class

- It is like Global.asax in the traditional .NET application. As the name suggests, it is executed first when the application starts.
- The startup class can be configured at the time of configuring the host in the Main() method of Program class.

Add a controller

- In Solution Explorer, right-click Controllers > Add > Controller
- In the Add Scaffold dialog box, select Controller Class – Empty
- In the Add Empty MVC Controller dialog, enter HelloWorldController and select Add.
- Replace the contents of Controllers/HelloWorldController.cs

```
public string Index() {  
    return "This is my default action...";  
}  
  
public string Welcome() {  
    return "This is the Welcome action method...";  
}
```

Add a controller

- MVC invokes controller classes (and the action methods within them) depending on the incoming URL.

- The default URL routing logic used by MVC uses a format like this:

`/[Controller]/[ActionName]/[Parameters]`

- The routing format is set in the Configure method in Startup.cs file.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

- Run your app & Check with these url in your browser

- <https://localhost:{PORT}/HelloWorld>
- <https://localhost:{PORT}/HelloWorld/Index>
- <https://localhost:{PORT}/HelloWorld/Welcome>

- Make changes for Welcome Method like this:

```
// GET: /HelloWorld/Welcome/
```

```
// Requires using System.Text.Encodings.Web;
```

```
public string Welcome(string name, int numTimes = 1)
```

```
{
```

```
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
```

```
}
```

- Check on your browser with these:

- <https://localhost:{PORT}/HelloWorld/Welcome?name=AAA&numtimes=4>

- Make changes again for Welcome Method with following code

```
public string Welcome(string name, int ID = 1) {  
    return HtmlEncoder.Default.Encode($"Hello {name}, ID is: {ID}");  
}
```

- Check on your browser with these:
 - <http://localhost:{PORT}/HelloWorld/Welcome/3?name=AAA>
- Here, the third URL segment matched the route parameter id. The Welcome method contains a parameter id that matched the URL template in the MapControllerRoute method in Startup.cs file. The trailing ? (in id?) indicates the id parameter is optional.

```
app.UseEndpoints(endpoints =>  
{  
    endpoints.MapControllerRoute(  
        name: "default",  
        pattern: "{controller=Home}/{action=Index}/{id?}");  
});
```


Add a view

- In your Project, Right click on the *Views* folder, and then **Add > New Folder** and name the folder *HelloWorld*.
- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item** dialog
 - In the search box in the upper-right, enter *view*
 - Select **Razor View**
 - Keep the **Name** box value, *Index.cshtml*.
 - Select **Add**
- Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following

```
@{  
    ViewData["Title"] = "Index";  
}  
  
<h2>Index</h2>  
  
<p>Hello from our View Template!</p>
```

Your Controller and View

```
public class HelloWorldController : Controller
{
    0 references
    public IActionResult Index()
    {
        return View();
    }
}
```

Index.cshtml

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

- Navigate to <https://localhost:{PORT}/HelloWorld>

Change views and layout pages

- In page, Select the menu links (WebApplicationCoreS1, Home, Privacy)
- Each page shows the same menu layout.
- The menu layout is implemented in the Views/Shared/_Layout.cshtml file.
- Open the Views/Shared/_Layout.cshtml file.
- Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site.
- Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, wrapped in the layout page.
- For example, if you select the Privacy link, the Views/Home/Privacy.cshtml view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

- Replace the content of the Views/Shared/_Layout.cshtml file with the following markup. The changes are highlighted:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - WebApplicationCoreS1</title>

<div class="container">
  <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">WebApplicationCoreS1</

<footer class="border-top footer text-muted">
  <div class="container">
    &copy; 2020 - WebApplicationCoreS1 - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</
```

Passing Data from the Controller to the View

- Controllers are responsible for providing the data required in order for a view template to render a response.
- In `HelloWorldController.cs`, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary.
- The `ViewData` dictionary is a dynamic object, which means any type can be used; the `ViewData` object has no defined properties until you put something inside it. The MVC model binding system automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method.
- Ex looks like :

Passing Data from the Controller to the View

```
public class HelloWorldController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult Welcome(string name, int numTimes = 1)
    {
        ViewData["Message"] = "Hello " + name;
        ViewData["NumTimes"] = numTimes;

        return View();
    }
}
```

Passing Data from the Controller to the View

```
@{
    ViewData["Title"] = "Welcome";
}

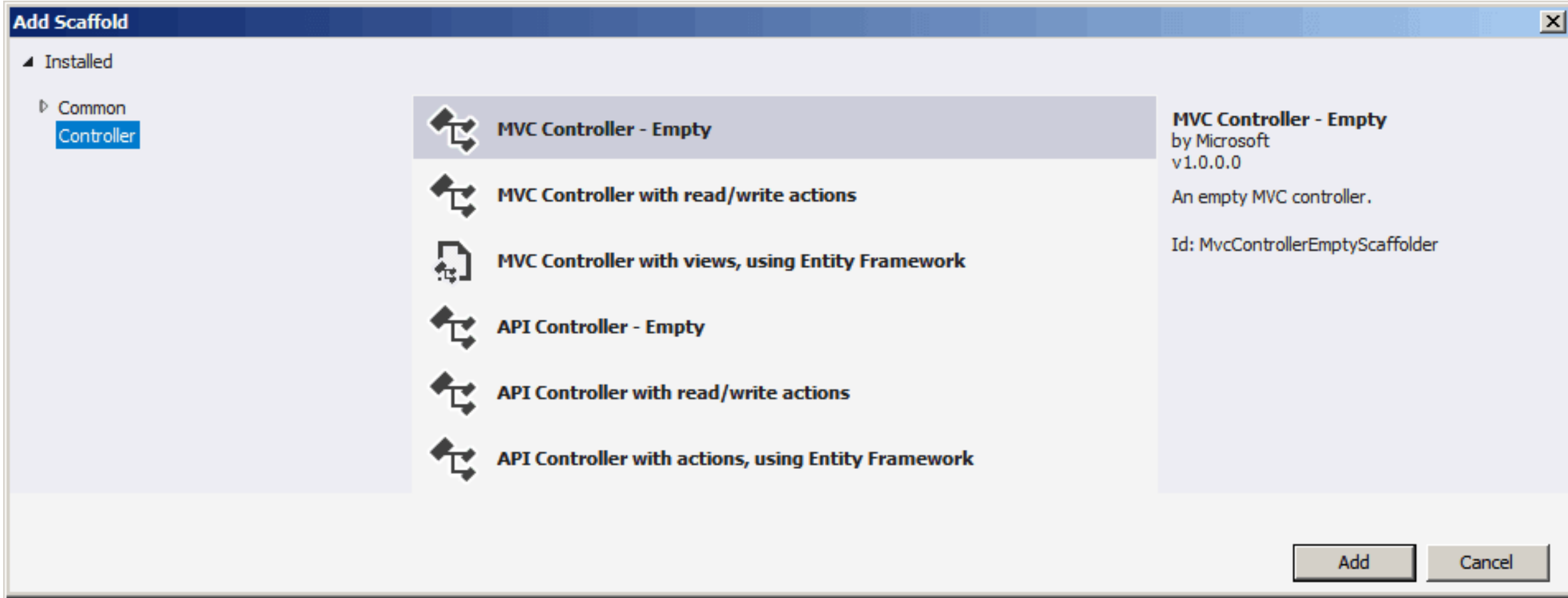
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Controllers Responsibilities

- Controllers are usually placed in a folder called "Controllers", directly in the root of your MVC project.
- They are usually named based on their purpose, with the word "Controller" as a suffix.
- The Controller has three major responsibilities
 - Handle the Request from the User
 - Build a Model – Controller Action method executes the application logic and builds a model
 - Send the Response – it returns the Result in HTML/File/JSON/XML or other format as requested by user.

Controller Scaffolding Option



Controller Scaffolding Option

- Both the MVC and API Controller inherits from the same Controller class and there is not much difference between them, except that API Controller is expected to return the data in serialized format to the client.
- Further, we have three options under both types of controllers.
 - Empty
 - With Read/Write Actions
 - With Views, using entity framework

Actions

- Controller is just a regular .NET class, it can have fields, properties and methods.
- Methods of a Controller class is referred to as **actions** - a method usually corresponds to an action in your application, which then returns something to the browser/user.
- All public methods on a Controller class is considered an Action.
- For instance, the browser might request a URL like /Products/Details/1 and then you want your ProductsController to handle this request with a method/action called Details.

When creating Action Method, points to remember

- Action methods Must be a public method
- The Action method cannot be a Static method or an Extension method.
- The Constructor, getter or setter cannot be used.
- Inherited methods cannot be used as the Action method.
- Action methods cannot contain ref or out parameters.
- Action Methods cannot contain the attribute [NonAction].
- Action methods cannot be overloaded

Actions Verbs

- To gain more control of how your actions are called, you can decorate them with the so-called Action Verbs.
- an action can be accessed using all possible HTTP methods (the most common ones are GET and POST)
- Edit action can be accessed with a GET request.

```
[HttpGet]
public IActionResult Edit()
{
    return View();
}
```

Actions Verbs

```
[HttpGet]
public IActionResult Edit()
{
    return Content("Edit");
}
```

```
[HttpPost]
public IActionResult Edit(Product product)
{
    product.Save();
    return Content("Product Updated!");
}
```

Actions Result Types

- When the Action (method) finishes its work, it will usually return something, as the `ActionResult` interface
- Some list of Action Result are:
 - `Content()` - returns specified string as plain text
 - `View()` - returns a View to the client
 - `PartialView()` - returns a Partial View to the client
 - `File()` - returns the content of a specified file to the client
 - `Json()` - returns a JSON response to the client
 - `Redirect()` and `RedirectPermanent()` - returns a redirect response to the browser (temporary or permanent), redirecting the user to another URL
 - `StatusCode()` - returns a custom status code to the client

Actions Result - Ex

- A common use case for this is to return either a View or a piece of content if the requested content is found, or a 404 (Page not Found) error if its not found. It could look like this:

```
public IActionResult Details(int id)
{
    Product product = GetProduct(id);
    if (product != null)
        return View(product);
    return NotFound();
}
```


Rendering HTML with Views

- In MVC pattern, the view handles the app's data presentation and user interaction.
- A view is an HTML template with embedded Razor markup. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client
- In ASP.NET Core MVC, views are .cshtml files that use the C# programming language in Razor markup. Usually, view files are grouped into folders named for each of the app's controllers. The folders are stored in a Views folder at the root of the app

Rendering HTML with Views

- The Home controller is represented by a Home folder inside the Views folder. The Home folder contains the views for the About, Contact, and Index (homepage) webpages. When a user requests one of these three webpages, controller actions in the Home controller determine which of the three views is used to build and return a webpage to the user.
- Use layouts to provide consistent webpage sections and reduce code repetition. Layouts often contain the header, navigation and menu elements, and the footer. The header and footer usually contain boilerplate markup for many metadata elements and links to script and style assets. Layouts help you avoid this boilerplate markup in your views.

Creating a View

- To create a view, add a new file and give it the same name as its associated controller action with the .cshtml file extension.
- For Ex - For About action in the Home controller, create an About.cshtml file in the Views/Home folder:

```
@{  
    ViewData["Title"] = "About";  
}  
<h2>@ViewData["Title"].</h2>  
<h3>@ViewData["Message"]</h3>  
<p>Use this area to provide additional information.</p>
```

Creating a View

- Razor markup starts with the @ symbol.
- You can write C# code within Razor code blocks set off by curly braces ({ ... }).
- You can display values within HTML by simply referencing the value with the @ symbol. See the contents of the <h2> and <h3> elements above.

How Controllers Specify Views

- Views are typically returned from actions as a `ViewResult`, which is a type of `ActionResult`.

HomeController.cs

```
public IActionResult About()
```

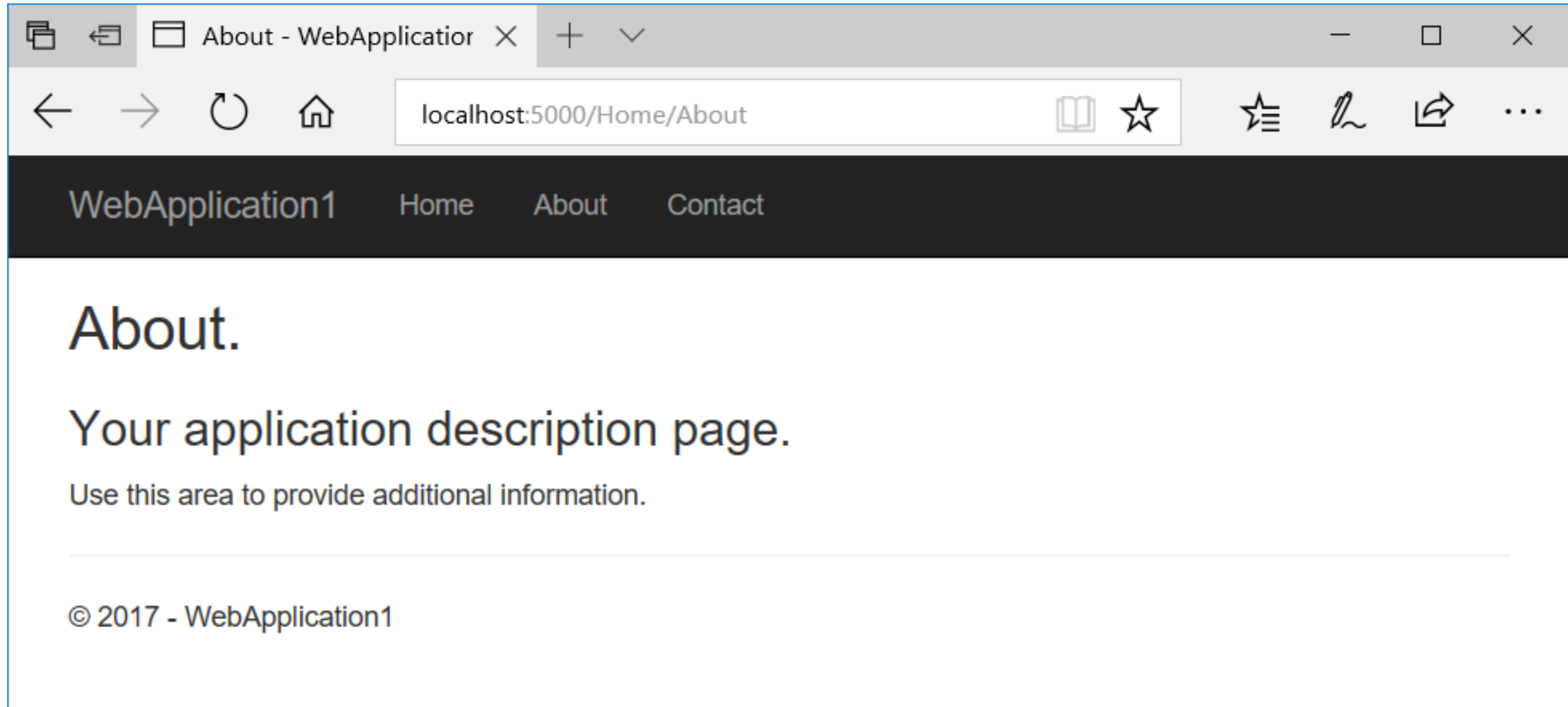
```
{
```

```
    ViewData["Message"] = "Your application description page.";
```

```
    return View();
```

```
}
```

How Controllers Specify Views



Passing Data to Views: ViewData & ViewBag

- Views have access to a weakly typed(loosely typed) collection of data. You can use these for passing small amounts of data in and out of controllers and views
- The ViewData property is a dictionary of weakly typed objects.
- The ViewBag property is a wrapper around ViewData that provides dynamic properties for the underlying ViewData collection.
- ViewData and ViewBag are dynamically resolved at runtime.
- ViewData is a ViewDataDictionary object accessed through string keys

Ex - ViewData

```
// HomeController.cs
```

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

```
    public IActionResult About() {
```

```
        ViewData["Message"] = "Your application description page.";
```

```
        return View();
```

```
    }
```

```
}
```

```
// About.cshtml
```

```
@{
```

```
    ViewData["Title"] = "About";
```

```
}
```

```
<h3>@ViewData["Message"]</h3>
```


Ex - ViewBag

```
// HomeController.cs
```

```
public class HomeController : Controller {  
    public IActionResult SomeAction() {  
        ViewBag.Greeting = "Hello";  
        return View();  
    }  
}
```

```
//SomeAction.cshtml
```

```
@{  
    ViewData["Title"] = "My Title";  
}  
<h3>@ViewBag.Greeting</h3>
```

Razor Syntax

- The biggest advantage of the Razor is the fact that you can mix client-side markup (HTML) with server-side code (e.g C# or VB.NET), without having to explicitly jump in and out of the two syntax types.
- In Razor, you can reference server-side variables etc. by simply prefixing it with an at-character (@).

`<p>Hello, the current date is: @DateTime.Now.ToString()</p>`

Ex - Razor & HTML Encoding

@{

```
var helloWorld = "<b>Hello, world!</b>";
```

}

<p>@helloWorld</p>

<p>@Html.Raw(helloWorld)</p>

Ex – Razor Explicit Expressions

@{

```
var name = "John Doe";
```

}

Hello, @(name.Substring(0,4)).

Your age is: @(37 + 5).

Ex – Multi-statement Razor blocks

```
@{  
    var sum = 32 + 10;  
    var greeting = "Hello, world!";  
    var text = "";  
    for(int i = 0; i <3; i++)  
    {  
        text += greeting + " The result is: " + sum + "\n";  
    }  
}
```

<h2>CodeBlocks</h2>

Text: @text

Razor Server-side Comments

- Sometimes you may want to leave comments in your code, or comment out lines of code temporarily to test things.

@*

Here's a Razor server-side comment

It won't be rendered to the browser

*@

Razor Server-side Comments

- If you're inside a Razor code block, you can even use the regular C# style comments:

```
@{  
    @*  
        Here's a Razor server-side comment  
    *@  
    // C# style single-line comment  
    /*  
    C# style multiline comment  
    It can span multiple lines  
    */  
}
```

Razor Syntax – Variables and Expressions

```
@{
```

```
    string helloWorldMsg = "Good day";
```

```
    if(DateTime.Now.Hour >17){
```

```
        helloWorldMsg = "Good evening";
```

```
    }
```

```
    helloWorldMsg += ", world!";
```

```
    helloWorldMsg = helloWorldMsg.ToUpper();
```

```
}
```

```
<div> @helloWorldMsg </div>
```


Razor Syntax – The if-else statement

```
@if(DateTime.Now.Year >= 2042)
{
    <span>The year 2042 has finally arrived!</span>
}
else
{
    <span>We're still waiting for the year of 2042...</span>
}
```

Razor Syntax – Loop

```
@{
    List<string> names = new List<string>() {
        "VB.NET", "C#", "Java"
    };
}
<ul>
    @for (int i = 0; i < names.Count; i++)
    {
        <li>@names[i]</li>
    }
</ul>
<ul>
@foreach (string name in names)
{
    <li>@name</li>
}
</ul>
```

Understanding Tag Helpers

- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. Tag helpers are a new feature and similar to HTML helpers, which help us render HTML.

Understanding Tag Helpers

- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. Tag helpers are similar to HTML helpers, which help us render HTML.
- There are many built-in Tag Helpers for common tasks, such as creating forms, links, loading assets etc.
- Tag Helpers are authored in C#, and they target HTML elements based on the element name, attribute name, or the parent tag.
- For ex, LabelTagHelper can target the HTML <label> element.
- Tag Helpers reduce the explicit transitions between HTML and C# in Razor views.

Understanding Tag Helpers

- In order to use Tag Helpers, we need to install a NuGet library and also add an `addTagHelper` directive to the view or views that use these tag helpers.
- Let us right-click on your project in the Solution Explorer and select Manage NuGet Packages....
- Search for `Microsoft.AspNet.Mvc.TagHelpers` and click the Install button.
- In the dependencies section, you will see "`Microsoft.AspNet.Mvc.TagHelpers`"

Understanding Tag Helpers

[Browse](#)

Installed

Updates

NuGet Package Manager: WebApplicationCoreS1

taghelpers



Include prerelease

Package source:

nuget.org



.NET

Microsoft.AspNetCore.Mvc.TagHelpers by Microsoft, **73.7M** downloads v2.2.0

ASP.NET Core MVC default tag helpers. Contains tag helpers for anchor tags, HTML input elements, caching, scripts, links (for CSS), and more.

.NET

Microsoft.AspNetCore.Razor by Microsoft, **85.5M** downloads v2.2.0

Razor is a markup syntax for adding server-side logic to web pages. This package contains runtime components for rendering Razor pages and implementing tag hel...

.NET

Microsoft.AspNetCore.Razor.Runtime by Microsoft, **85.3M** downloads v2.2.0

Runtime infrastructure for rendering Razor pages and tag helpers.



Localization.AspNetCore.TagHelpers by AdmiringWorm, **151K** downloads v0.6.0

Asp.Net Core Tag Helpers to use when localizing Asp.Net Core applications instead of manually injecting IViewLocator.

Microsoft.AspNetCore.Mvc.TagHelpers

Version: Latest stable 2.2.0

Install

Options

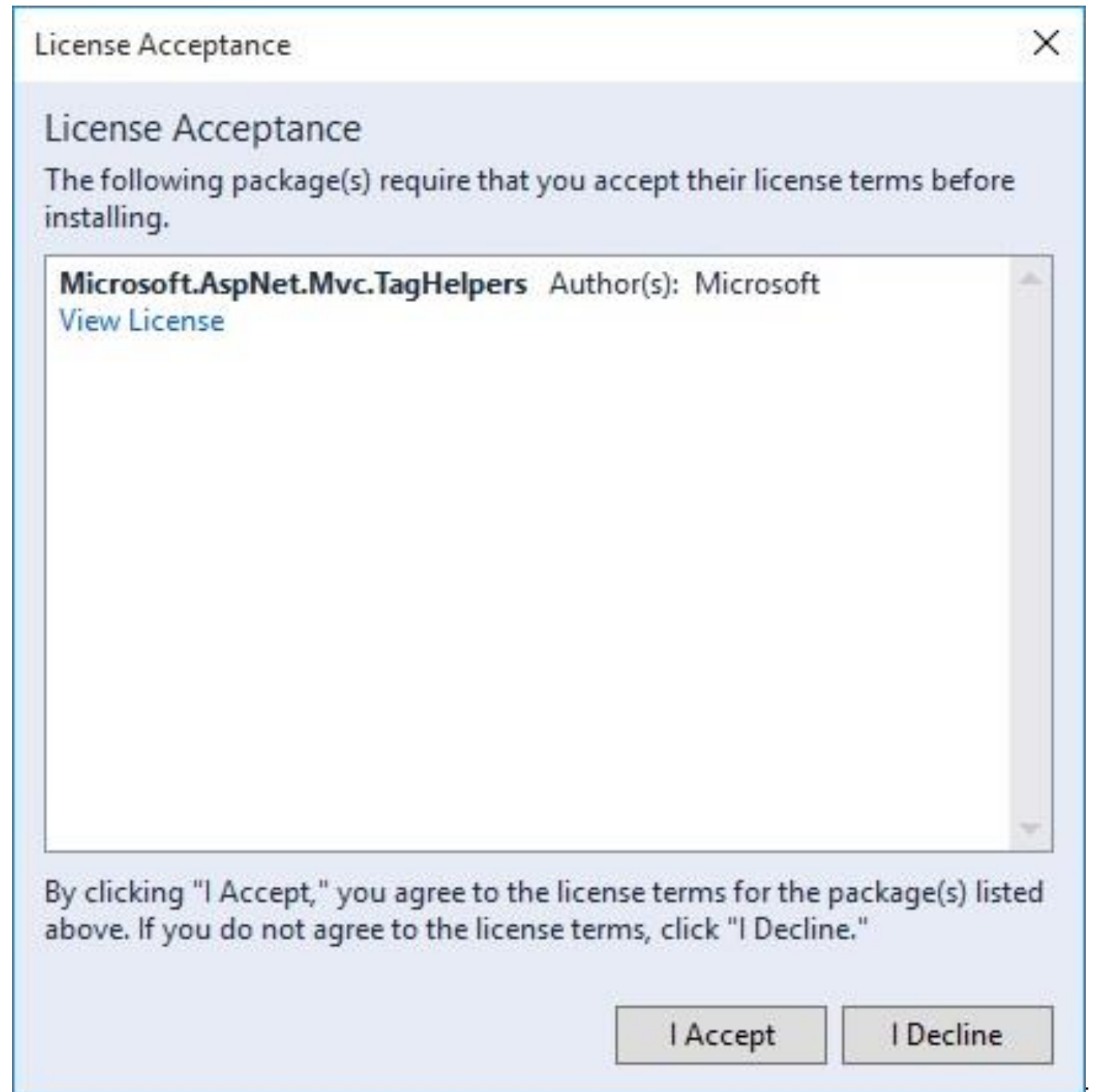
Description

ASP.NET Core MVC default tag helpers. Contains tag helpers for anchor tags, HTML input elements, caching, scripts, links (for CSS), and more.

This package was built from the source code at <https://github.com/aspnet/Mvc/tree/a6199bbfbab05583f987bae322fb04566841aaea>

Version: 2.2.0

- You will receive the following dialog box.
- Click on I Accept



Writing your own Tag Helpers

- You can also write your own tag helper. You can place it right inside your application project, but you need to tell the Razor view engine about the tag helper. By default, they are not just rendered down to the client, even though these tag helpers look like they blend into the HTML.
- Razor will call into some code to process a tag helper; it can remove itself from the HTML and it can also add additional HTML.
- You need to register your tag helpers with Razor, even the Microsoft tag helpers, in order for Razor to be able to spot these tag helpers in the markup and to be able to call into the code that processes the tag helper.
- The directive to do that is `addTagHelper`, and you can place this into an individual view or `ViewImports` file.

Form Tag Helper

- The Form Tag Helper is bound to the HTML `<form>` element.
- provides several server-side attributes which help us to manipulate the generated HTML.
- Some of the available attributes are
 - **asp-controller:** The name of the MVC controller to use
 - **asp-action:** The name of the MVC Controller action method to use
 - **asp-area:** The name of the Controller Area to use

Form Tag Helper

EX

```
<form asp-controller="Home" asp-action="Create">
```

The above code translated into

```
<form method="post" action="/Home/Create">  
  <input name="__RequestVerificationToken" type="hidden"  
  value="CfDJ8P1Iso5McDB0jgPkvG904mnNiAE8U0HkV1A9e-  
  Mtc76u7fSjCnoy909Co49eGlbyJx  
  pp-nYphF_Xk0rPo0tTGdygc2H8nCtZCcGURMZ9Uf01fP0g5jRARxTHXnb8N6yYADtdQSn  
  JItXtYsir8GCWqZM" />  
</form>
```

Form Tag Helper

Label tag Helper :

```
<label asp-for="@Model.Name"></label>
```

Which translates into `<label for="Name">Name</label>`

- Using @Model keyword is optional here. You can directly use the model property name as shown below.

```
<label asp-for="Name"></label>
```

Input Tag Helper - Similarly, the Input tag Helper is applied to the input HTML element.

```
<input asp-for="Name"/>
```

Which translates into `<input type="text" id="Name" name="Name" value="" />`

- The type, id & name attributes are automatically derived from the *model property type & data annotations* applied on the model property

Form Tag Helper

EX

```
<form asp-controller="Home" asp-action = "Create">  
  <label asp-for = "Name"></label>  
  <input asp-for = "Name"/>  
  <label asp-for = "Rate"></label>  
  <input asp-for = "Rate"/>  
  <label asp-for = "Rating"></label>  
  <input asp-for = "Rating"/>  
  <input type="submit" name="submit"/>  
</form>
```

List of Built-in Tag Helpers

TagHelper	Targets	Attributes
Form Tag Helper	<Form>	asp-action, asp-all-route-data, asp-area, asp-controller, asp-protocol, asp-route, asp-route-
Anchor Tag Helpers	<a>	asp-action, asp-all-route-data, asp-area, asp-controller, asp-Protocol, asp-route, asp-route-
Image Tag Helper		append-version
Input Tag Helper	<input>	for
Label Tag Helper	<label>	For
Link Tag Helper	<link>	href-include, href-exclude, fallback-href, fallback-test-value, append-version

List of Built-in Tag Helpers

TagHelper	Targets	Attributes
Options Tag Helper	<select>	asp-for, asp-items
Partial Tag Helper	<partial>	name, model, for, view-data
Script Tag Helper	<script>	src-include, src-exclude, fallback-src, fallback-src-include, fallback-src-exclude fallback-test, append-version
Select Tag Helper	<select>	for, items
Textarea Tag Helper	<textarea>	for
Validation Message Tag Helper		validation-for
Validation Summary Tag Helper	<div>	validation-summary

Model

- The Model in an MVC application should represent the current state of the application, as well as business logic and/or operations.
- A very important aspect of the MVC pattern is the Separation of Concerns (SoC). SoC is achieved by encapsulating information inside a section of code, making each section modular, and then having strict control over how each module communicates. For the MVC pattern, this means that both the View and the Controller can depend on the Model, but the Model doesn't depend on neither the View nor the Controller.
- As mentioned, the Model can be a class already defined in your project, or it can be a new class you add specifically to act as a Model. Therefore, Models in the ASP.NET MVC framework usually exists in a folder called "Models".

ViewModels

- There are, however, a lot of situations where you may want to create a specific ViewModel for a specific View. This can be to extend or simplify an existing Model, or because you want to represent something in a View that's not already covered by one of your models.
- ViewModels are often placed in their own directory in your project, called "ViewModels".
- Some people also prefer to postfix the name of the class with the word ViewModel, e.g. "AddressViewModel" or "EditUserViewModel".

When to use ViewModel?

- **To represent something in a View that's not already contained by an existing Model:** When you pass a Model to a View, you are free to pass e.g. a String or another simple type, but if you want to pass multiple values, it might make more sense to create a simple ViewModel to hold the data, like this one:

```
public class AddressViewModel
{
    public string StreetName { get; set; }
    public string ZipCode { get; set; }
}
```

When to use ViewModel?

- **To access the data of multiple Models from the same View:** This is relevant in a lot of situations, e.g. when you want to create a FORM where you can edit the data of multiple Models at the same time. You could then create a ViewModel like this:

```
public class EditItemsViewModel
{
    public Model1Type Model1 { get; set; }
    public Model2Type Model2 { get; set; }
}
```

When to use ViewModel?

- **To simplify an existing Model:** Imagine that you have a huge class with information about a user. Perhaps even sensitive information like passwords. When you want to expose this information to a View, it can be beneficiary to only expose the parts of it you actually need. For instance, you may have a small widget showing that the user is logged in, which username they have and for how long they have been logged in. So instead of passing your entire User Model, you can pass in a much leaner ViewModel, designed specifically for this purpose:

```
public class SimpleUserInfoViewModel {  
    public string Username { get; set; }  
    public TimeSpan LoginDuration { get; set; }  
}
```

When to use ViewModel?

- **To extend an existing Model with data only relevant to the View:** On the other hand, sometimes your Model contains less information than what you need in your View. An example of this could be that you want some convenience properties or methods which are only relevant to the View and not your Model in general, like in this example where we extend a user Model (called WebUser) with a LoginDuration property, calculated from the LastLogin DateTime property already found on the WebUser class:

```
public class WebUser
{
    public DateTime LastLogin { get; set; }
}
```

From there on there are two ways of doing things: You can either extend this class (inherit from it) or add a property for the WebUser instance on the ViewModel. Like this:

```
public class UserInfoViewModel {  
    public WebUser User { get; set; }  
    public TimeSpan LoginDuration  
    {  
        get { return DateTime.Now - this.User.LastLogin; }  
    }  
}
```

Or like this:

```
public class ExtendedUserInfoViewModel : WebUser {  
    public TimeSpan LoginDuration  
    {  
        get { return DateTime.Now - this.LastLogin; }  
    }  
}
```

Model Binding in ASP.NET Core

- The Model Binding extracts the data from an HTTP request and provides them to the controller action method parameters. The action method parameters may be simple types like integers, strings, etc. or complex types such as Student, Order, Product, etc.
- The controller action method handle the incoming HTTP Request.
- Our application default route template ({controller=Home}/{action=Index}/{Id?})
- When you load this url - <http://localhost:52191/home/details/101>, shows the Details action method.

```
public IActionResult Details(int Id)
{
    var studentDetails = listStudents.FirstOrDefault(std => std.StudentId == Id);
    return View(studentDetails);
}
```

Using Model Binding

- In Model Folder, create a class *WebUser.cs*

```
public class WebUser
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

- In Controller Folder, create a new Controller as *UserController.cs* and add action method as follows:

```
[HttpGet]
public IActionResult SimpleBinding()
{
    return View(new WebUser() { FirstName = "John", LastName = "Doe" });
}
```

- By letting our View know what kind of Model it can expect, with the @model directive, we can now use various helper methods (more about those later) to help with the generation of a FORM:
- In View Folder, create a file *SimpleBinding.cshtml*

```
@using(var form = Html.BeginForm())
{
    <div>
        @Html.LabelFor(m => m.FirstName)
        @Html.TextBoxFor(m => m.FirstName)
    </div>

    <div>
        @Html.LabelFor(m => m.LastName)
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <input type="submit" value="Submit" />
}
```


- The result will be a very generic-looking FORM, but with labels and textboxes designated to host the properties of your Model:

FirstName	<input type="text" value="John"/>
LastName	<input type="text" value="Doe"/>
<input type="submit" value="Submit"/>	

- By default, the FORM will be posted back to the same URL that delivered it, so to handle that, we need a POST-accepting Controller method to handle the FORM submission:

```
[HttpPost]
public IActionResult SimpleBinding(WebUser webUser)
{
    //TODO: Update in DB here...
    return Content($"User {webUser.FirstName} updated!");
}
```

Data Annotations

- Data Annotations (sometimes referred to as Model Attributes), which basically allows you to add meta data to a property.
- The cool thing about DataAnnotations is that they don't disturb the use of your Models outside of the MVC framework.
- When generating the label, the name of the property is used, but property names are generally not nice to look at for humans. As an example of that, we might want to change the display-version of the FirstName property to "First Name".

```
public class WebUser
{
    [Display(Name="First Name")]
    public string FirstName { get; set; }
}
```

Model Validation

- They will allow you to enforce various kinds of rules for your properties, which will be used in your Views and in your Controllers, where you will be able to check whether a certain Model is valid in its current state or not (e.g. after a FORM submission).
- Let's add just a couple of basic validation to the WebUser

```
public class WebUser {  
    [Required]  
    [StringLength(25)]  
    public string FirstName { get; set; }  
    [Required]  
    [StringLength(50, MinLength(3))]  
    public string LastName { get; set; }  
    [Required]  
    [EmailAddress]  
    public string MailAddress { get; set; }  
}
```

Model Validation

- Notice how the three properties have all been decorated with DataAnnotations.
- First of all, all properties have been marked with the [Required] attribute, meaning that a value is required - it can't be NULL.
- [StringLength] attribute make requirements about the maximum, and in one case minimum, length of the strings. These are of course particularly relevant if your Model corresponds to a database table, where strings are often defined with a maximum length.
- For the last property, [EmailAddress] attribute ensure that the value provided looks like an e-mail adress.

```
@model HelloMVCWorld.Models.WebUser
@using(var form = Html.BeginForm()) {
    <div>
        @Html.LabelFor(m => m.FirstName)
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div>
        @Html.LabelFor(m => m.LastName)
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <div>
        @Html.LabelFor(m => m.MailAddress)
        @Html.TextBoxFor(m => m.MailAddress)
    </div>
    <input type="submit" value="Submit" />
}
```

In your View Page

```
public class ValidationController : Controller
{
    [HttpGet]
    public IActionResult SimpleValidation()
    {
        return View();
    }
    [HttpPost]
    public IActionResult SimpleValidation(WebUser webUser)
    {
        if(ModelState.IsValid)
            return Content("Thank you!");
        else
            return Content("Model could not be validated!");
    }
}
```

In your Controller

- In POST action, we check the IsValid property of the ModelState object. Depending on the data submitted in the FORM, it will be either true or false, based on the validation rules we defined for the Model (WebUser).
- With this in place, you can now prevent a Model from being saved, e.g. to a database, unless it's completely valid.

```

@model HelloMVCWorld.Models.WebUser
@using(var form = Html.BeginForm()) {
    <div>
        @Html.LabelFor(m => m.FirstName)
        @Html.TextBoxFor(m => m.FirstName)
        @Html.ValidationMessageFor(m => m.FirstName)
    </div>
    <div>
        @Html.LabelFor(m => m.LastName)
        @Html.TextBoxFor(m => m.LastName)
        @Html.ValidationMessageFor(m => m.LastName)
    </div>
    <div>
        @Html.LabelFor(m => m.MailAddress)
        @Html.TextBoxFor(m => m.MailAddress)
        @Html.ValidationMessageFor(m => m.MailAddress)
    </div>
    <input type="submit" value="Submit" />
}

```

Displaying validation errors

- Let's extend our FORM so that it can display error messages to the user. We can use helper method `ValidationMessageFor()`.
- It will simply output the error message related to the field, if there is one - otherwise, nothing will be outputted. Here's the extended version of our FORM:

```
public class ValidationController : Controller
{
    [HttpGet]
    public IActionResult SimpleValidation()
    {
        return View();
    }
    [HttpPost]
    public IActionResult SimpleValidation(WebUser webUser)
    {
        if(ModelState.IsValid)
            return Content("Thank you!");
        else
            return View(webUser);
    }
}
```

In your Controller

- make sure that once the FORM is submitted, and if there are validation errors, we return the FORM to the user, so that they can see and fix these errors.
- We do that in our Controller, simply by returning the View and the current Model state, if there are any validation errors:

Try Submitting the Form

- Try submitting the FORM with empty fields, you should be immediately returned to the FORM, but with validation messages next to each of the fields.

FirstName The FirstName field is required.
LastName The LastName field is required.
MailAddress The MailAddress field is required.

- If you try submitting the FORM with a value that doesn't meet the StringLength requirements, you will notice that there are even automatically generated error messages for these as well. For instance, if you submit the FORM with a LastName that's either too long or too short, you will get this message:
- The field LastName must be a string with a minimum length of 3 and a maximum length of 50.

What if you want more control of these messages?

- But No problem, they can be overridden directly in the DataAnnotations of the Model. Here's a version of our Model where we have applied custom error messages:

```
public class WebUser {
    [Required(ErrorMessage = "You must enter a value for the First Name field!")]
    [StringLength(25, ErrorMessage = "The First Name must be no longer than 25
characters!")]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "You must enter a value for the Last Name field!")]
    [StringLength(50, MinimumLength = 3, ErrorMessage = "The Last Name must be between 3
and 50 characters long!")]
    public string LastName { get; set; }

    [Required(ErrorMessage = "You must enter a value for the Mail Address field!")]
    [EmailAddress(ErrorMessage = "Please enter a valid e-mail address!")]
    public string MailAddress { get; set; }
}
```

```

@model HelloMVCWorld.Models.WebUser
@using(var form = Html.BeginForm())
{
    @Html.ValidationSummary()
    <div>
        @Html.LabelFor(m => m.FirstName)
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div>
        @Html.LabelFor(m => m.LastName)
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <div>
        @Html.LabelFor(m => m.MailAddress)
        @Html.TextBoxFor(m => m.MailAddress)
    </div>
    <input type="submit" value="Submit" />
}

```

Displaying a validation summary

- You must enter a value for the First Name field!
- The Last Name must be between 3 and 50 characters long!
- Please enter a valid e-mail address!

FirstName
 LastName aa
 MailAddress test

- Use ValidationSummary() method found on the Html helper object:
- Now, When the FORM is submitted, It will be returned with validation errors

Types of Model Validation DataAnnotations

- **[Required]** - Specifies that a value needs to be provided for this property
- **[StringLength]** - Allows you to specify at least a maximum amount of characters. We can also add Minimum Length as well.

```
[StringLength(50, MinimumLength = 3)]
```

- **[Range]** - specify a minimum and a maximum value for a numeric property (int, float, double etc.)

```
[Range(1, 100)]
```

- **[Compare]** - allows you to set up a comparison between the property

```
[Compare("MailAddressRepeated")]
```

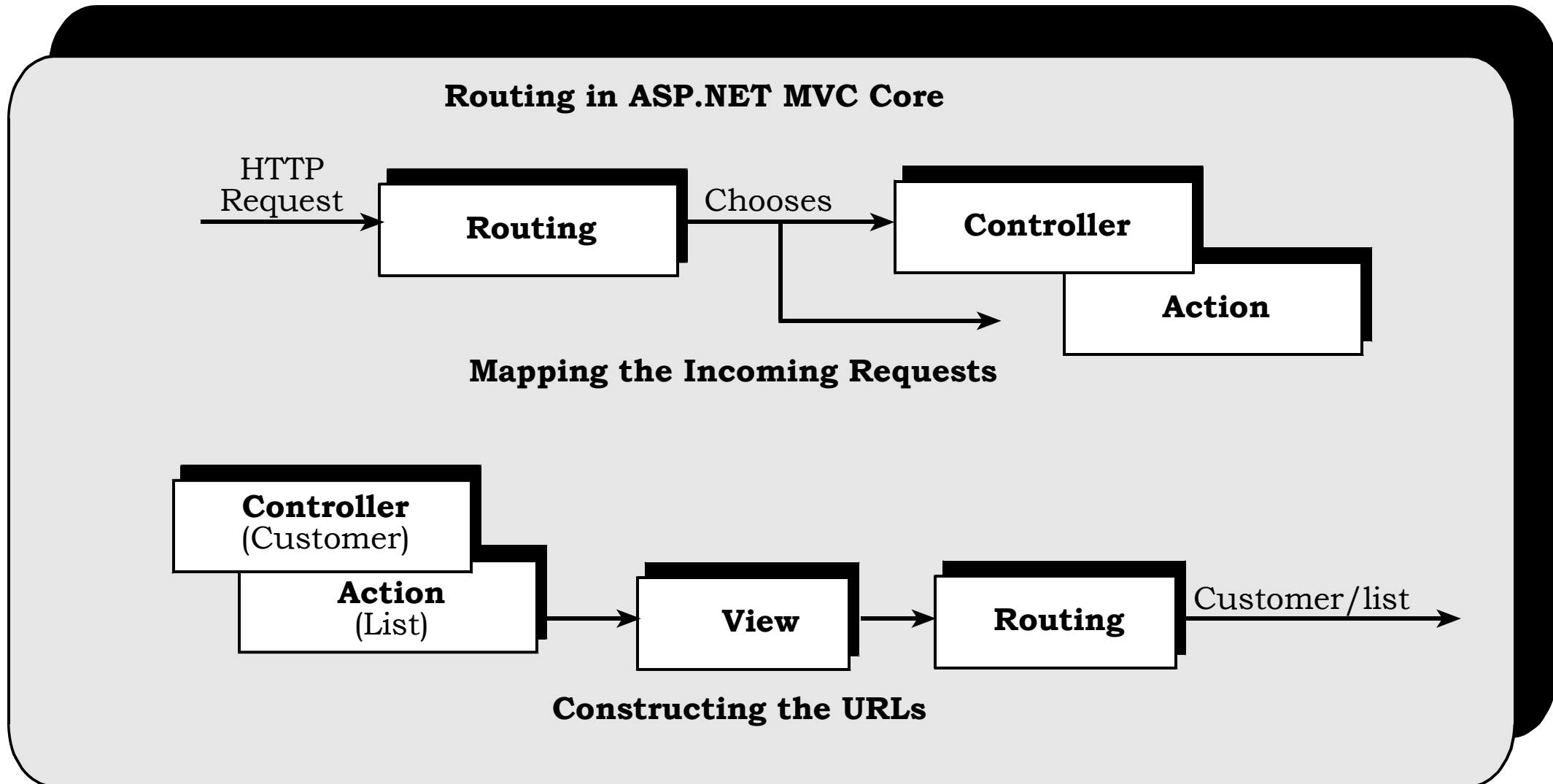
```
public string MailAddress { get; set; }
```

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

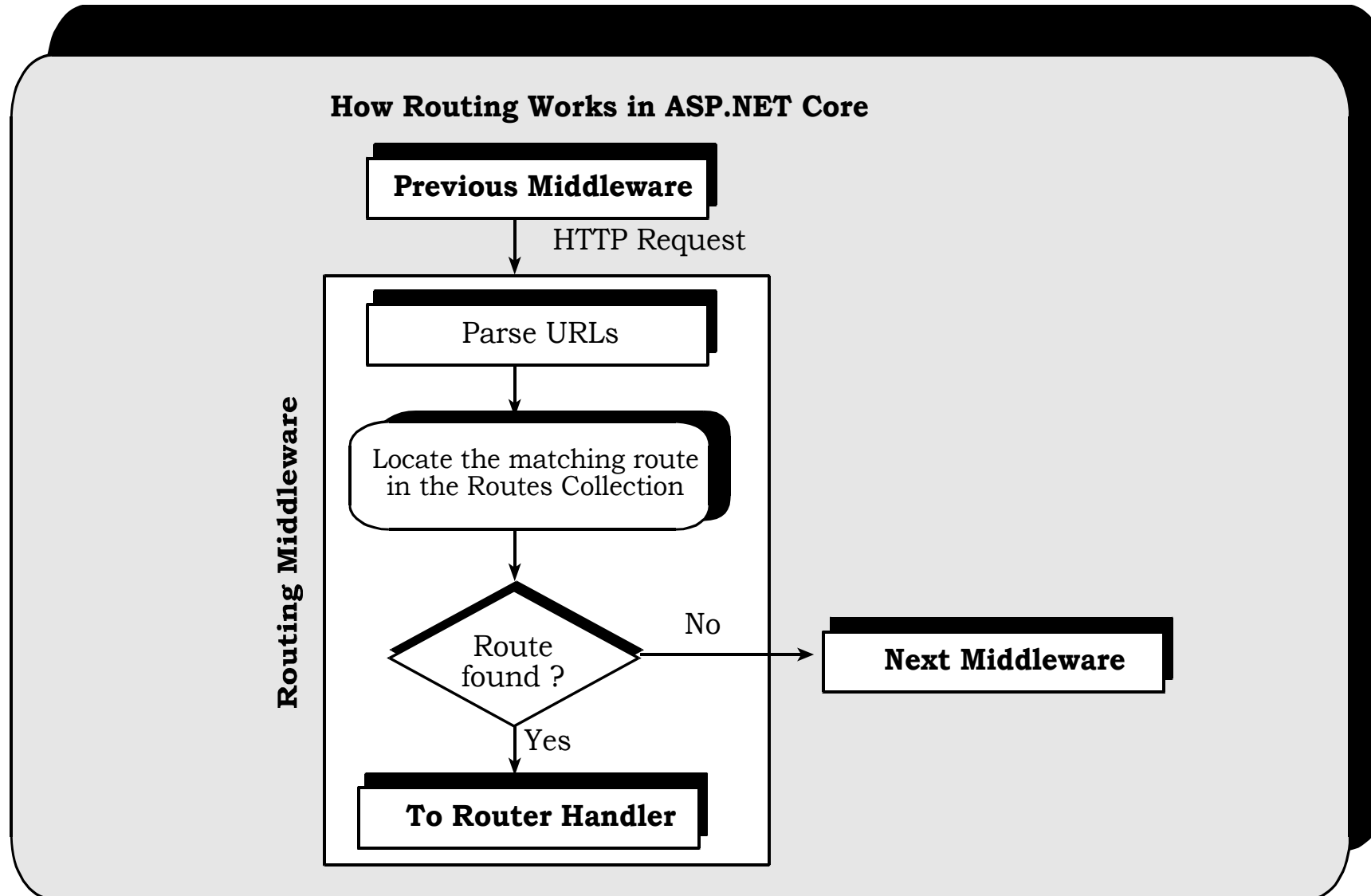
URL Routing and Features

- The Routing is the Process by which ASP.NET Core inspects the incoming URLs and maps them to Controller Actions.
- It also used to generate the outgoing URLs.
- This process is handled by the Routing Middleware. The Routing Middleware is available in `Microsoft.AspNetCore.Routing` Namespace .
- The Routing has two main responsibilities:
 1. It maps the incoming requests to the Controller Action
 2. Generate an outgoing URLs that correspond to Controller actions.

Routing in ASP.NET MVC Core



How Routing works in ASP.NET MVC Core



How Routing works in ASP.NET MVC Core

- When the Request arrives at the Routing Middleware it does the following.
 1. It Parses the URL.
 2. Searches for the Matching Route in the RouteCollection.
 3. If the Route found then it passes the control to RouteHandler.
 4. If Route not found, it gives up and invokes the next Middleware.

How Routing works in ASP.NET MVC Core

What is a Route

- The Route is similar to a roadmap. We use a roadmap to go to our destination. Similarly, the ASP.NET Core Apps uses the Route to go to the controller action.
- The Each Route contains a Name, URL Pattern (Template), Defaults and Constraints. The URL Pattern is compared to the incoming URLs for a match. An example of URL Pattern is `{controller=Home}/{action=Index}/{id?}`
- The Route is defined in the `Microsoft.AspNetCore.routing` namespace .

How Routing works in ASP.NET MVC Core

What is a Route Collection

- The Route Collection is the collection of all the Routes in the Application.
- An app maintains a single in-memory collection of Routes. The Routes are added to this collection when the application starts.
- The Routing Module looks for a Route that matches the incoming request URL on each available Route in the Route collection.
- The Route Collection is defined in the namespace `Microsoft.AspNetCore.routing`.

How Routing works in ASP.NET MVC Core

What is a Route Handler

- The Route Handler is the Component that decides what to do with the route.
- When the routing Engine locates the Route for an incoming request, it invokes the associated RouteHandler and passes the Route for further processing. The Route handler is the class which implements the IRouteHandler interface.
- In the ASP.NET Core, the Routes are handled by the MvcRouteHandler.

How Routing works in ASP.NET MVC Core

MVCRouteHandler

- Default Route Handler for the ASP.NET Core MVC Middleware. The MVCRouteHandler is registered when we register the MVC Middleware in the Request Pipeline. You can override this and create your own implementation of the Route Handler.
- defined in the namespace `Microsoft.AspNetCore.Mvc`.
- The MVCRouteHandler is responsible for invoking the Controller Factory, which in turn creates the instance of the Controller associated the Route.
- The Controller then takes over and invokes the Action method to generate the View and Complete the Request.

How to setup Routes

- There are two different ways by which we can set up routes.
 1. Convention-based routing
 2. Attribute routing

Convention-based routing

- The Convention based Routing creates routes based on a series of conventions, defined in the ASP.NET Core Startup.cs file.

Attribute routing

- Creates routes based on attributes placed on controller actions.

The two routing systems can co-exist in the same system.

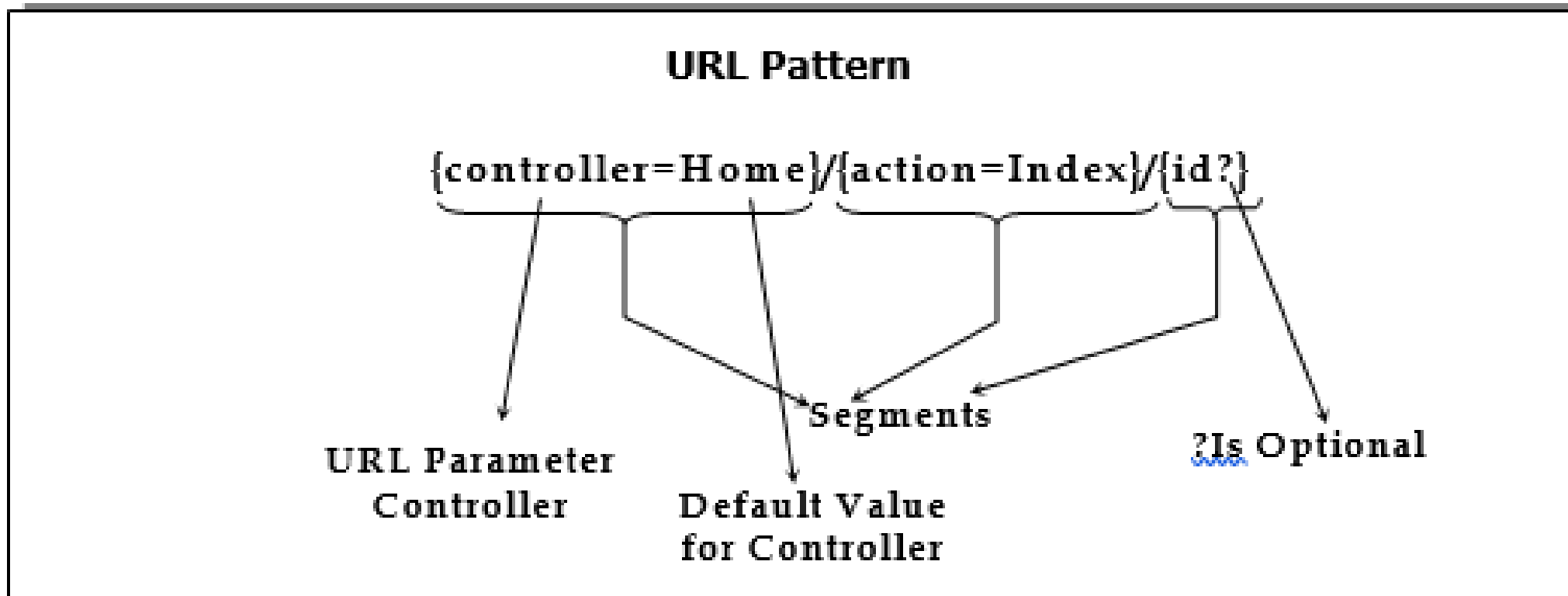
How to setup Routes

- The Convention based Routes are configured in the Configure method of the Startup class. The Routing is handled by the Router Middleware. ASP.NET MVC adds the routing Middleware to the Middleware pipeline when using the `app.UseMvc` or `app.UseMvcWithDefaultRoute`.

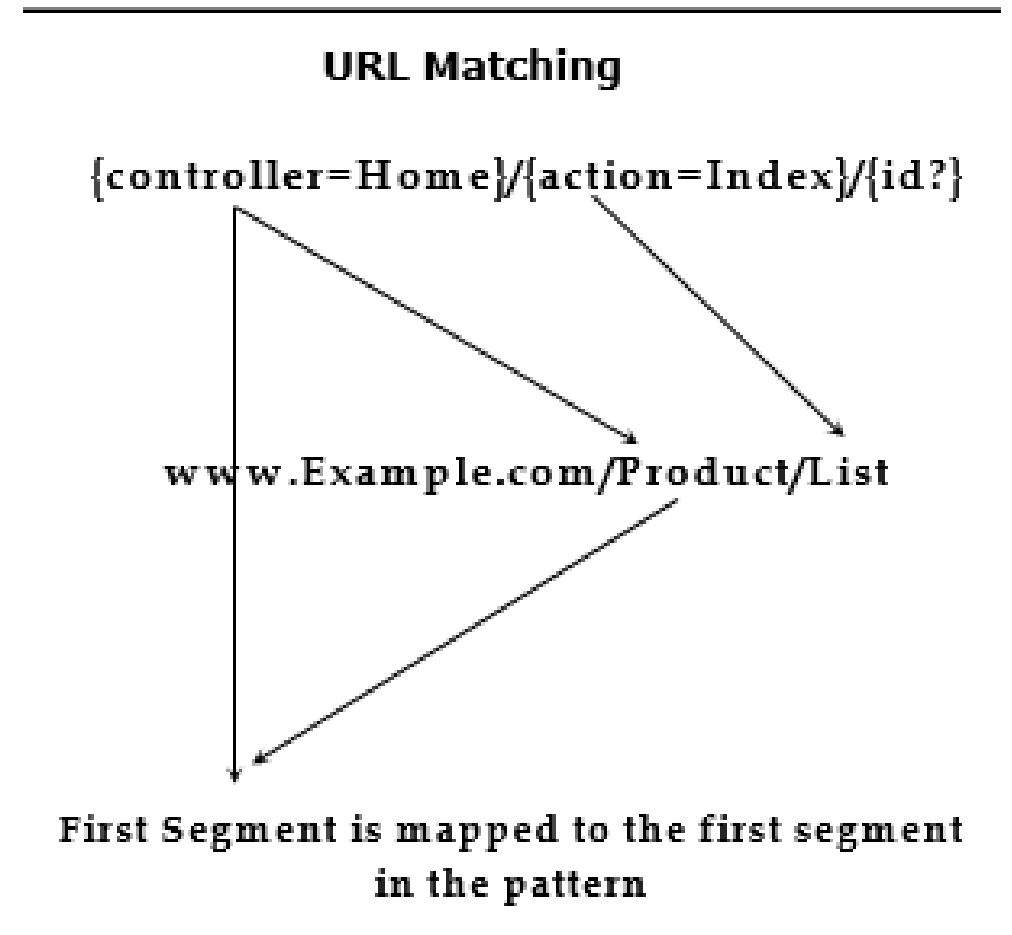
```
MapRoute Api
app.UseMvc(routes=>
{
    routes.MapRoute(
        "default",           → Name of the Route
        "{controller=Home}/{action=Index}/{id?}"), → URL
        Pattern
    }
```

URL Patterns

- The Each route must contain a URL pattern. This Pattern is compared to an incoming URL. If the pattern matches the URL, then it is used by the routing system to process that URL.



- The URL Pattern `{controller=Home}/{action=Index}/{id?}` Registers route where the first part of the URL is Controller, the second part is the action method to invoke on the controller. The third parameter is an additional data in the name of id.
- The Each segment in the incoming URL is matched to the corresponding segment in the URL Pattern.
- `{controller=Home}/{action=Index}/{id?}` has three segments. The last one is optional.

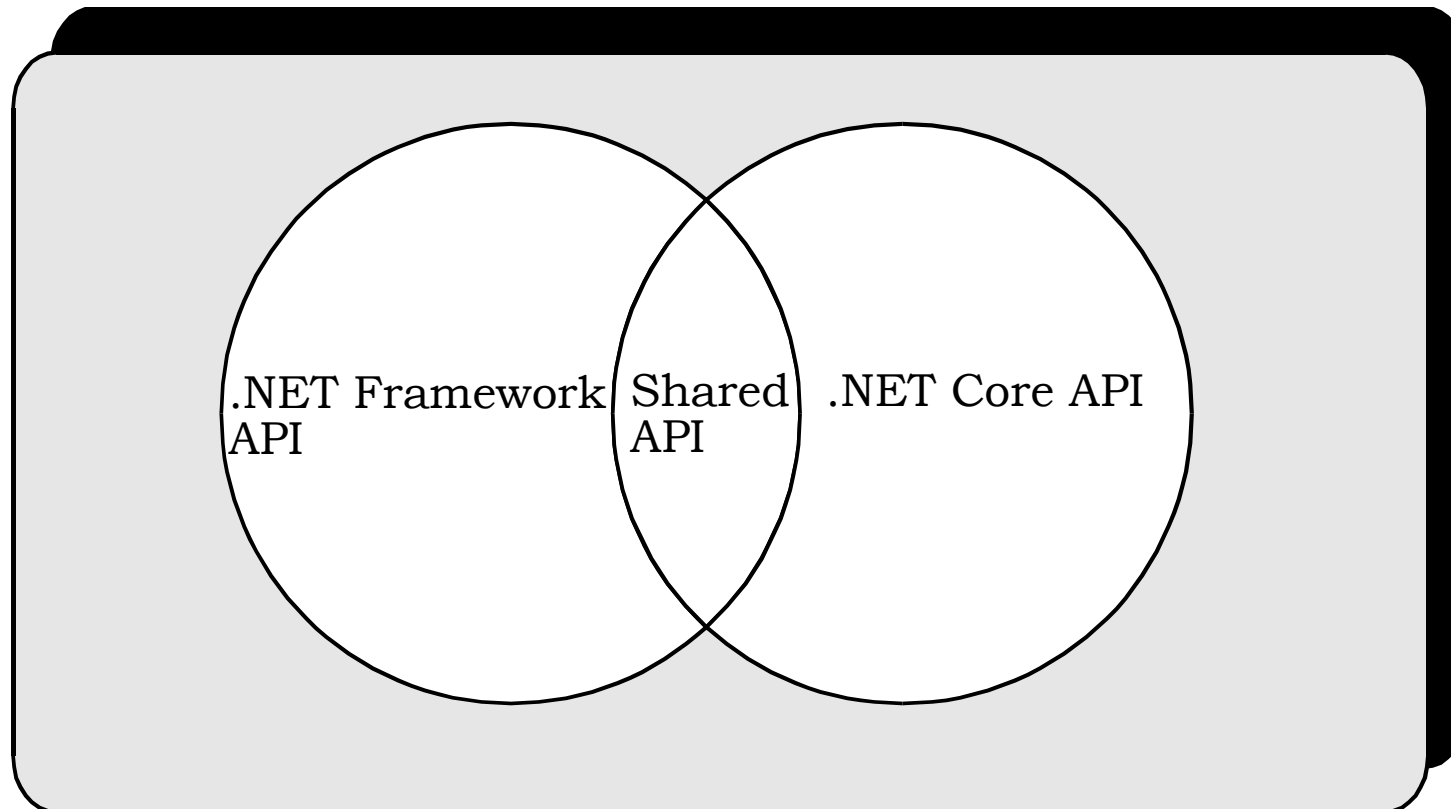


Web API Applications

- Before ASP.NET Web API core, the two-different framework MVC and Web API were pretty much similar.
- Both used to support Controller and action methods. In earlier version, the main purpose of Web API was to make REST API calls and there were view engine like Razor.
- On the other hand, MVC was designed for HTML front ends to communicate to backend in a standard a web application. However, when ASP.NET Web API core was released, the main target was to support JSON based REST API. It combines the key feature of both MVC and old Web API framework.

ASP.NET Core Web API Architecture

- ASP.NET Web API is mainly based on the MVC architecture. The .NET framework and .NET Core also share a number of APIs.



New Features in ASP.NET Core Web API

- **Cross Platform** - ASP.NET Web API Core is cross-platform; therefore, it is suitable for running on any platform like Windows, Mac, or Linux. Earlier ASP.NET applications were not able to run on Linux and Mac operating system.
- **Backward Compatibility** - For existing application, ASP.NET Web API Core supports two framework.
- **Faster** - ASP.NET Web API Core is much faster than previous versions
- **Static Content** - wwwroot folder contain all the static content e.g. js, css, images.

Creating Web API in ASP.NET Core

- Create the controller that have 3 things:
 - should have [ApiController] attribute on them. This attribute tells that the controller will serve HTTP API Responses.
 - derive from ControllerBase class instead of Controller class.
 - should have attribute routing applied on them like [Route("someUrl/[controller]")].
 - The controller of a Web API looks like:

```
[ApiController]
[Route("someURL/[controller]")]
public class ExampleController : ControllerBase
```

API Controllers

- API Controller is just a normal Controller, that allows data in the model to be retrieved or modified, and then deliver it to the client. It does this without having to use the actions provided by the regular controllers.
- The data delivery is done by following a pattern known by name as REST. REST Stands for REpresentational State Transfer pattern, which contains 2 things:
 - Action Methods which do specific operations and then deliver some data to the client. These methods are decorated with attributes that makes them to be invoked only by HTTP requests.
 - URLs which defines operational tasks. These operations can be – sending full or part of a data, adding, deleting or updating records. In fact it can be anything.

API Controller

- MVC and API controllers both derive from the Controller class, which derives from ControllerBase:

```
public class MyMvc20Controller : Controller {}  
[Route("api/[controller]")]  
public class MyApi20Controller : Controller {}
```

- As of Core 2.1 (and 2.2), the template-generated classes look a little different, where a Web controller is a child of the Controller class and an API controller is a child of ControllerBase.

```
public class MyMvc21Controller : Controller {}  
[Route("api/[controller]")]  
public class MyApi21Controller : ControllerBase {}
```

API Controller

- This can be expressed in the table below:

Namespace	Microsoft.AspNetCore.Mvc
Common parent	ControllerBase (Abstract Class)
MVC Controller parent	Controller: ControllerBase
MVC Controller	MyMvcController: Controller

JSON

- The new built-in JSON support, `System.Text.Json`, is high-performance, low allocation, and based on `Span<byte>`.
- The `System.Text.Json` namespace provides high-performance, low-allocating, and standards-compliant capabilities to process JavaScript Object Notation (JSON), which includes serializing objects to JSON text and deserializing JSON text to objects, with UTF-8 support built-in.
- It also provides types to read and write JSON text encoded as UTF-8, and to create an in-memory document object model (DOM) for random access of the JSON elements within a structured view of the data.

Adding JSON Patch To Your ASP.Net Core Project

- Run Package Manager and install JSON Patch Library with command:
 - `Install-Package Microsoft.AspNetCore.JsonPatch`

- Write in your controller

```
[Route("api/[controller]")]
public class PersonController : Controller {
    private readonly Person _defaultPerson = new Person
    {
        FirstName="Jim",
        LastName="Smith"
    };
    [HttpPatch("update")]
    public Person Patch([FromBody]JsonPatchDocument<Person> personPatch) {
        personPatch.ApplyTo(_defaultPerson);
        return _defaultPerson;
    }
}
```

```
public class Person
{
    public string FirstName{get;set;}
    public string LastName{get;set;}
}
```

Adding JSON Patch To Your ASP.net Core Project

- In above example we are just using a simple object stored on the controller and updating that, but in a real API we will be pulling the data from a datasource, applying the patch, then saving it back.

- When we call this endpoint with the following payload :

```
[{"op": "replace", "path": "FirstName", "value": "Bob"}]
```

- We get the response of :

```
{"firstName": "Bob", "lastName": "Smith"}
```

first name got changed to Bob!

DEPENDENCY INJECTION AND IOC CONTAINERS

- ASP.NET Core is designed from scratch to support Dependency Injection.
- ASP.NET Core injects objects of dependency classes through constructor or method by using built-in IoC container.
- ASP.NET Core framework contains simple out-of-the-box IoC container which does not have as many features as other third party IoC containers. If you want more features such as auto-registration, scanning, interceptors, or decorators then you may replace built-in IoC container with a third party container.

BUILT-IN IOC CONTAINER

- The built-in container is represented by `IServiceProvider` implementation that supports constructor injection by default. The types (classes) managed by built-in IoC container are called services.
- There are basically two types of services in ASP.NET Core:
 1. Framework Services: Services which are a part of ASP.NET Core framework such as `IApplicationBuilder`, `IHostingEnvironment`, `ILoggerFactory` etc.
 2. Application Services: The services (custom types or classes) which you as a programmer create for your application.
- In order to let the IoC container automatically inject our application services, we first need to register them with IoC container.

Registering Application Service

- Consider the following example of simple ILog interface and its implementation class. We will see how to register it with built-in IoC container and use it in our application.

```
public interface ILog {  
    void info(string str);  
}  
  
class MyConsoleLogger : ILog {  
    public void info(string str)  
    {  
        Console.WriteLine(str);  
    }  
}
```

Registering Application Service

- ASP.NET Core allows us to register our application services with IoC container, in the ConfigureServices method of the Startup class. The ConfigureServices method includes a parameter of IServiceCollection type which is used to register application services.
- Let's register above ILog with IoC container in ConfigureServices() method as shown below. Example: Register Service

```
public class Startup {  
    public void ConfigureServices(IServiceCollection services) {  
        services.Add(new ServiceDescriptor(typeof(ILog),  
            new MyConsoleLogger()));  
    } // other code removed for clarity..  
}
```

Registering Application Service

- In above ex:
- Add() method of IServiceCollection instance is used to register a service with an IoC container.
- ServiceDescriptor is used to specify a service type and its instance. We have specified ILog as service type and MyConsoleLogger as its instance. This will register ILog service as a singleton by default.
- Now, an IoC container will create a singleton object of MyConsoleLogger class and inject it in the constructor of classes wherever we include ILog as a constructor or method parameter throughout the application.
- Thus, we can register our custom application services with an IoC container in ASP.NET Core application. There are other extension methods available for quick and easy registration of services.

Understanding Service Lifetime for Registered Service

- Built-in IoC container manages the lifetime of a registered service type. It automatically disposes a service instance based on the specified lifetime.
- The built-in IoC container supports three kinds of lifetimes:
 1. Singleton: IoC container will create and share a single instance of a service throughout the application's lifetime.
 2. Transient: The IoC container will create a new instance of the specified service type every time you ask for it.
 3. Scoped: IoC container will create an instance of the specified service type once per request and will be shared in a single request.

Understanding Service Lifetime for Registered Service

- The following example shows how to register a service with different lifetimes.
- Example: Register a Service with Lifetime

```
public void ConfigureServices(IServiceCollection services)
{
    // singleton
    services.Add(new ServiceDescriptor(typeof(ILog), new MyConsoleLogger()));
    services.Add(new ServiceDescriptor(typeof(ILog), typeof(MyConsoleLogger),
        ServiceLifetime.Transient)); // Transient
    services.Add(new serviceDescriptor(typeof(ILog), typeof(MyConsoleLogger),
        ServiceLifetime.Scoped)); // Scoped
}
```

IOC Containers

- ASP.NET Core framework includes built-in IoC container for automatic dependency injection. The built-in IoC container is a simple yet effective container.
- The followings are important interfaces and classes for built-in IoC container:

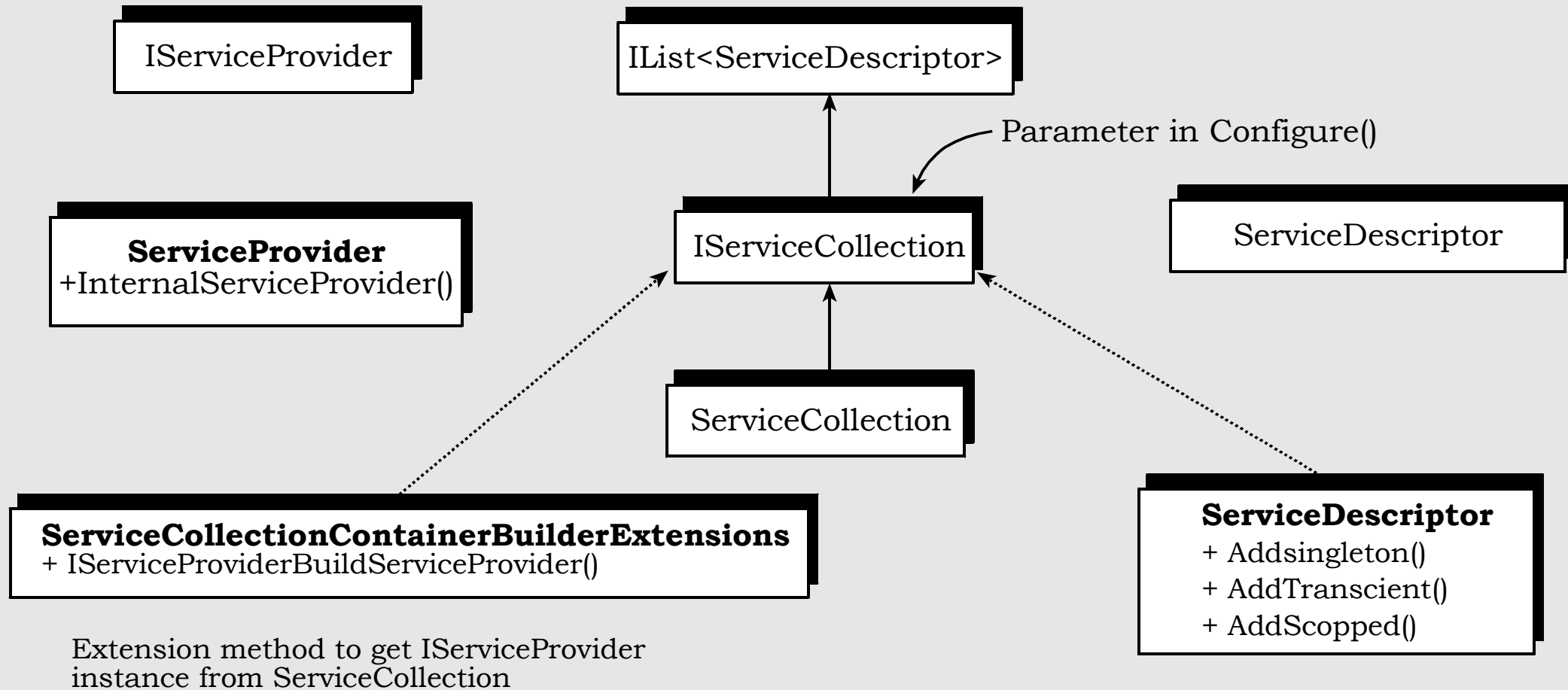
Interfaces

1. IServiceProvider
2. IServiceCollection

Classes

1. ServiceProvider
2. ServiceCollection
3. ServiceDescription
4. ServiceCollectionServiceExtensions
5. ServiceCollectionContainerBuilderExtensions

IOC Containers



IOC Containers

IServiceCollection

- we can register application services with built-in IoC container in the Configure method of Startup class by using IServiceCollection. IServiceCollection interface is an empty interface. It just inherits IList<servicedescriptor>.
- The ServiceCollection class implements IServiceCollection interface.
- So, the services you add in the IServiceCollection type instance, it actually creates an instance of ServiceDescriptor and adds it to the list.

IServiceProvider

- IServiceProvider includes GetService method.
- The ServiceProvider class implements IServiceProvider interface which returns registered services with the container. We cannot instantiate ServiceProvider class because its constructors are marked with internal access modifier.

IOC Containers

ServiceCollectionServiceExtensions

- The ServiceCollectionServiceExtensions class includes extension methods related to service registrations which can be used to add services with lifetime. AddSingleton, AddTransient, AddScoped extension methods defined in this class.

ServiceCollectionContainerBuilderExtensions

- ServiceCollectionContainerBuilderExtensions class includes BuildServiceProvider extension method which creates and returns an instance of ServiceProvider.
- There are three ways to get an instance of IServiceProvider:
 - Using IApplicationBuilder
 - Using HttpContext
 - Using IServiceCollection

IOC Containers

Using IApplicationBuilder

- We can get the services in Configure method using IApplicationBuilder's ApplicationServices property as shown below.

```
public void Configure(IServiceProvider pro, IApplicationBuilder app,
    IHostingEnvironment env)
{
    var services = app.ApplicationServices;
    var logger = services.GetService<ILog>() }
    //other code removed for clarity
}
```

IOC Containers

Using HttpContext

```
var services = HttpContext.RequestServices;  
var log = (ILog)services.GetService(typeof(ILog));
```

Using IServiceCollection

```
public void ConfigureServices(IServiceCollection services)  
{  
    var serviceProvider = services.BuildServiceProvider();  
}
```