Unit 4

Creating ASP.NET Core MVC Applications

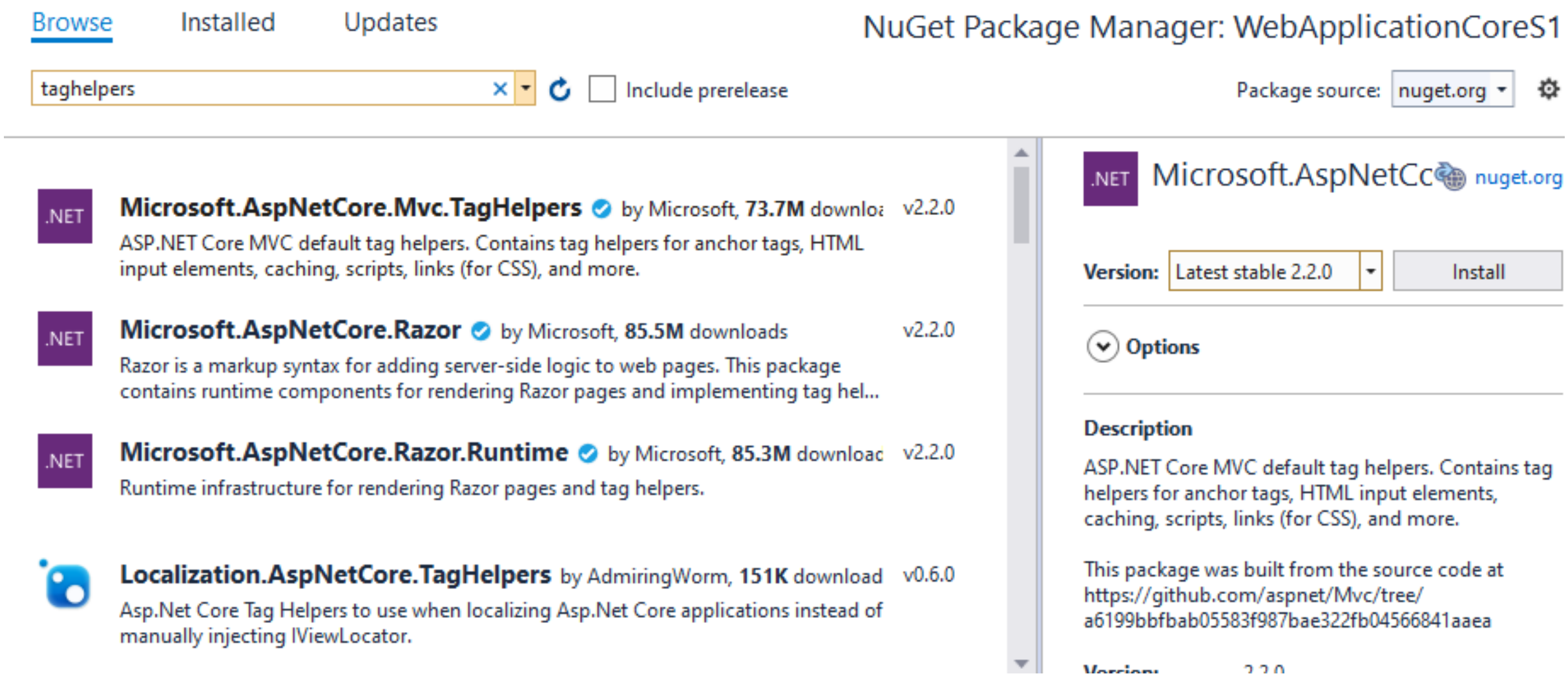
Understanding Tag Helpers

- 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.

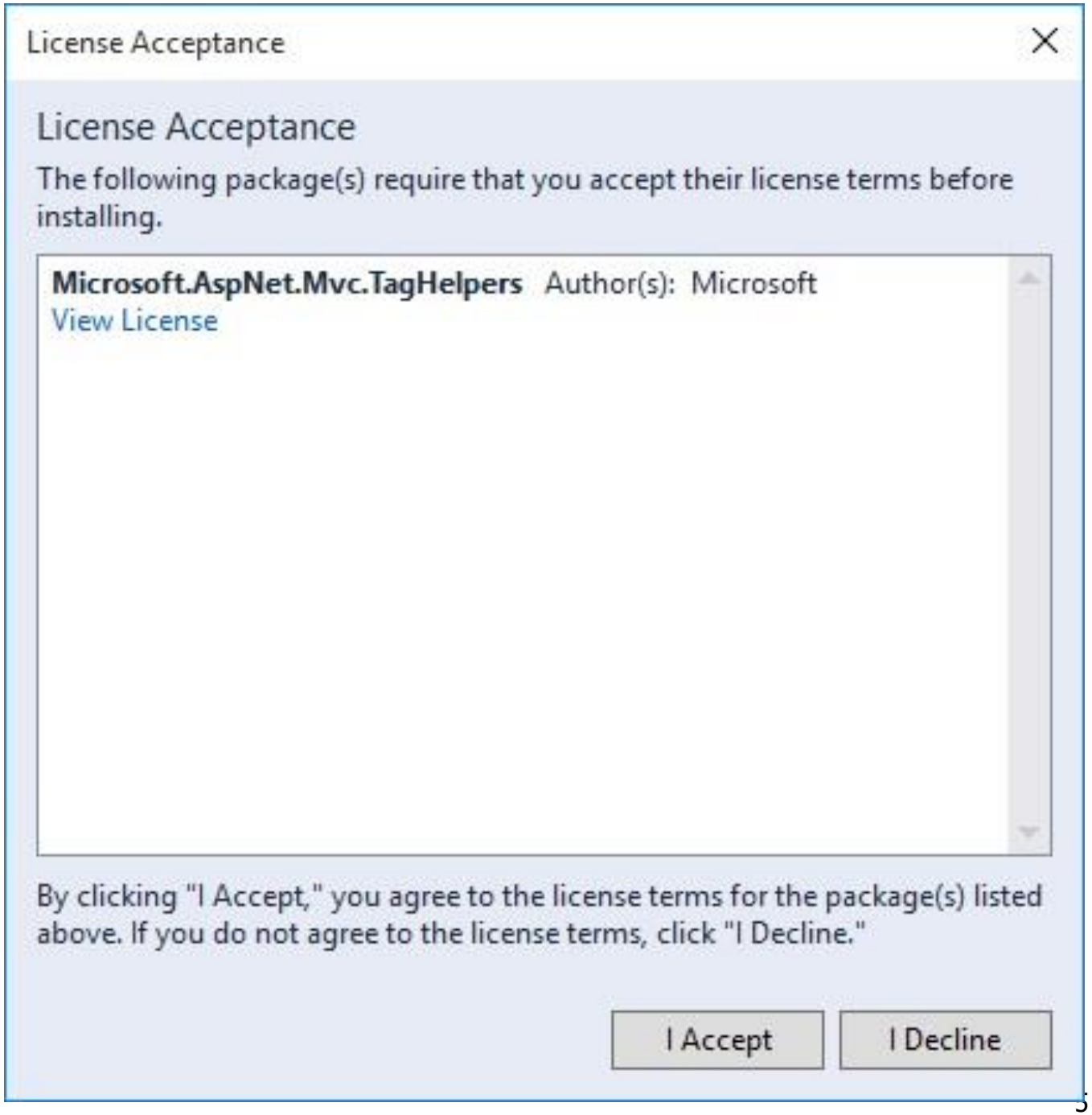
Understanding Tag Helpers

- 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



- 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="CfDJ8PlIso5McDBOjgPkVg9O4mnNiAE8U0HkVlA9e-
  Mtc76u7fSjCnoy909Co49eGlbyJx
  pp-nYphF_XkOrPo0tTGdygc2H8nCtZCcGURMZ9Uf01fPOg5jRARxTHXnb8N6yYADtdQSn
  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	<img>	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		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 ViewResult 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:

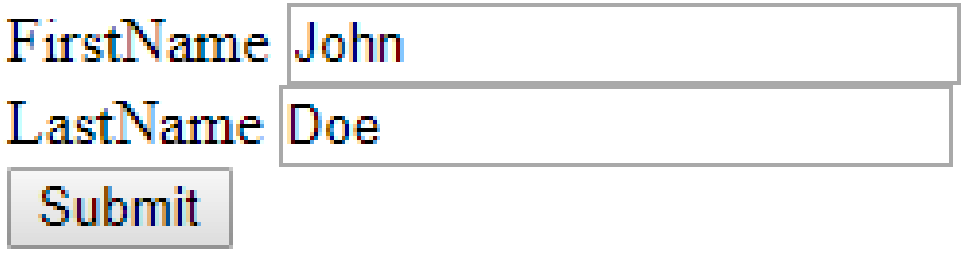
```
[HttpGet]
public IActionResult SimpleBinding()
{
    return View(new WebUser() { FirstName = "John", LastName = "Doe" });
}
```

- 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:



- 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:

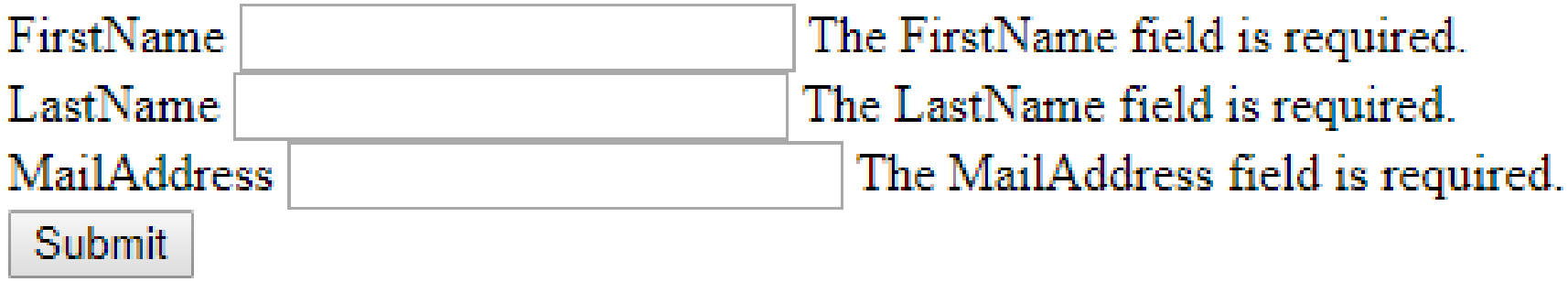
```
public class ValidationController : Controller
{
    [HttpGet]
    public IActionResult SimpleValidation()
    {
        return View();
    }
    [HttpPost]
  public IActionResult SimpleValidation(WebUser webUser)
  {
      if(ModelState.IsValid)
          return Content("Thank you!");
      else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



- 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
Submit

- 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 Minumum 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)]
```

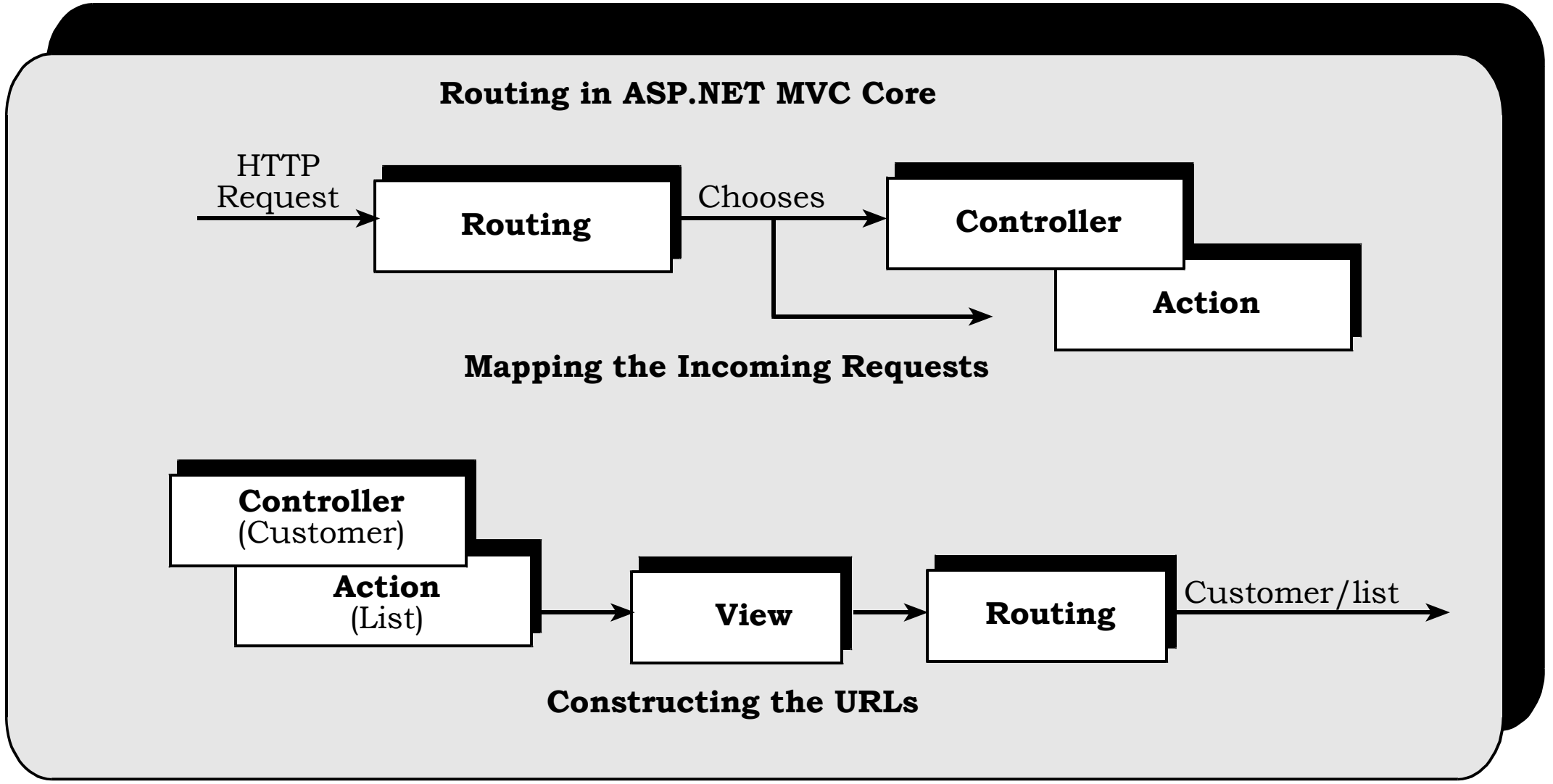
- **[Compare] -** allows you to set up a comparison between the property

```
[Compare("MailAddressRepeated")]
public string MailAddress { get; set; }
public string MailAddressRepeated { get; set; }
```

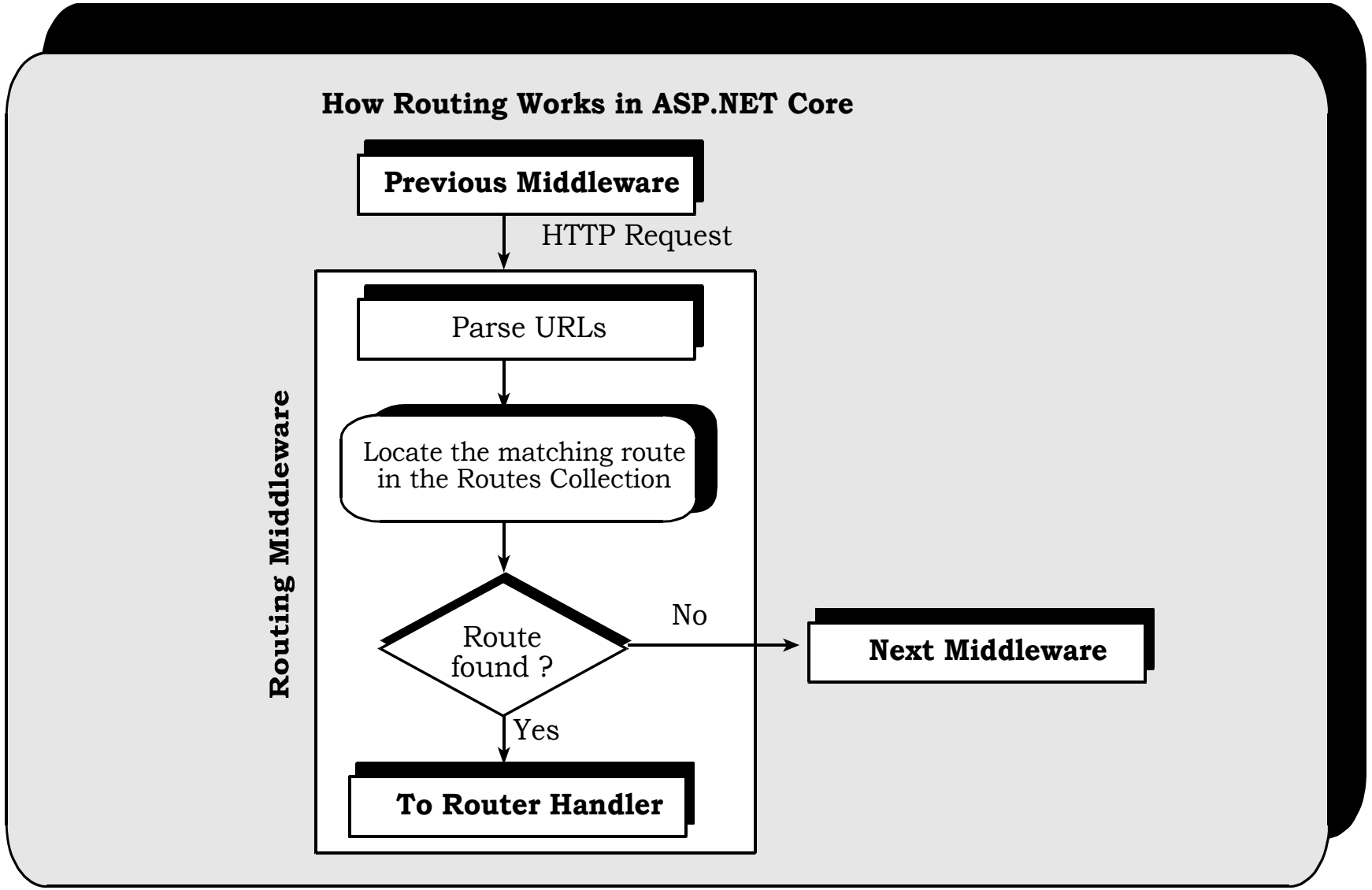
URL Routing and Features

- 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

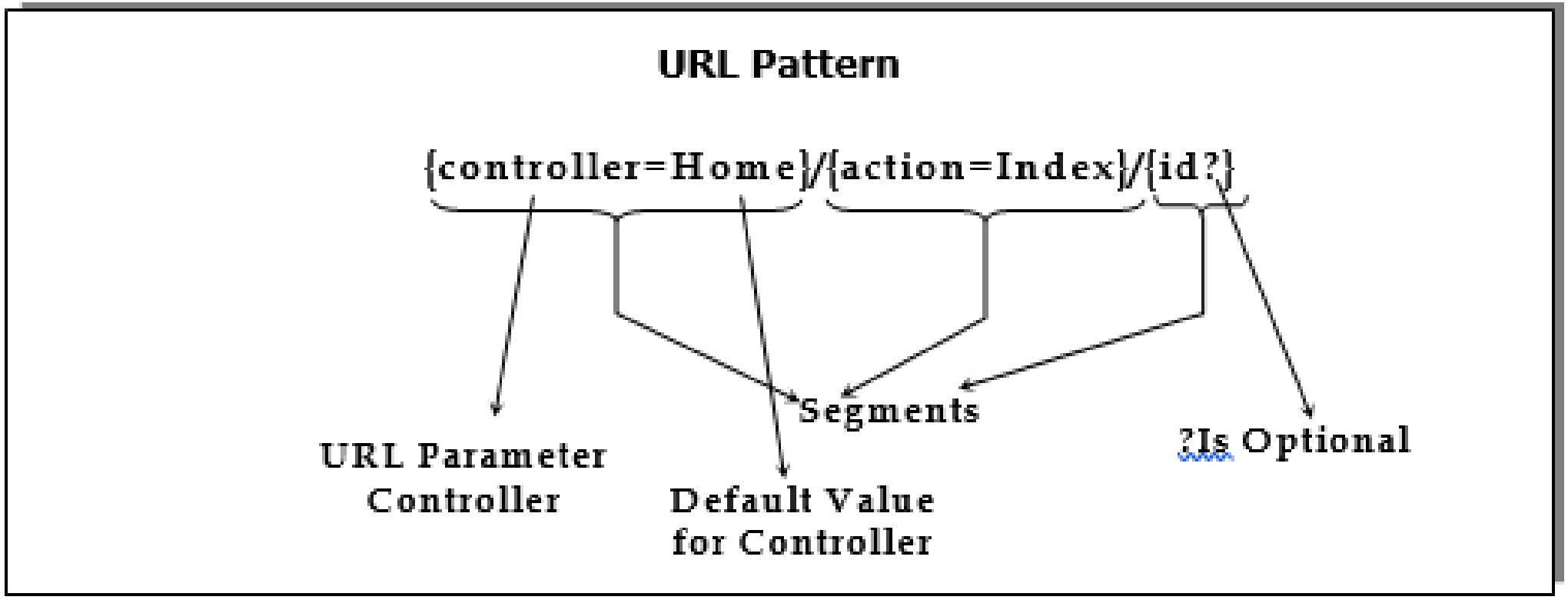
- The Convention based Routes are configured in the Configure method of the Startup class. The Routing is handled by the Router Middleware. ASP.NET MVC adds the routing Middleware to the Middleware pipeline when using the app.UseMVC or app.UseMvcWithDefaultRoute.

MapRoute Api

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

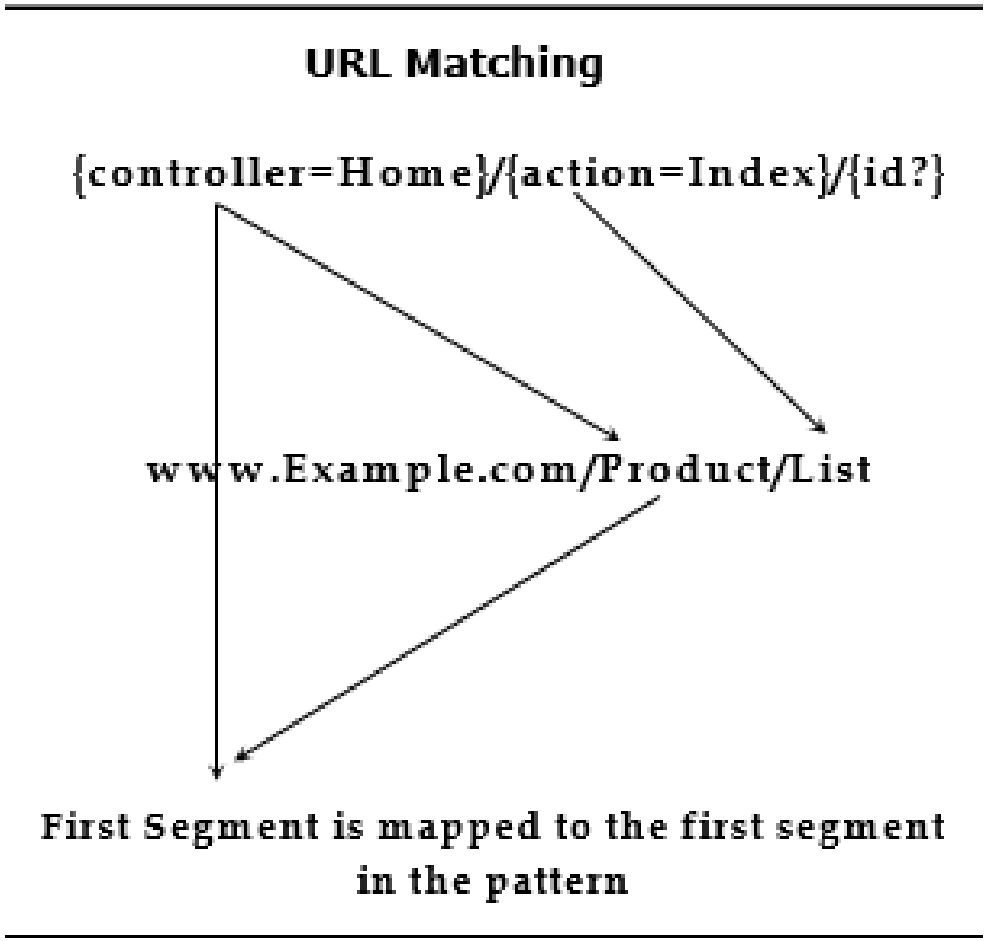
URL Patterns

- 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.

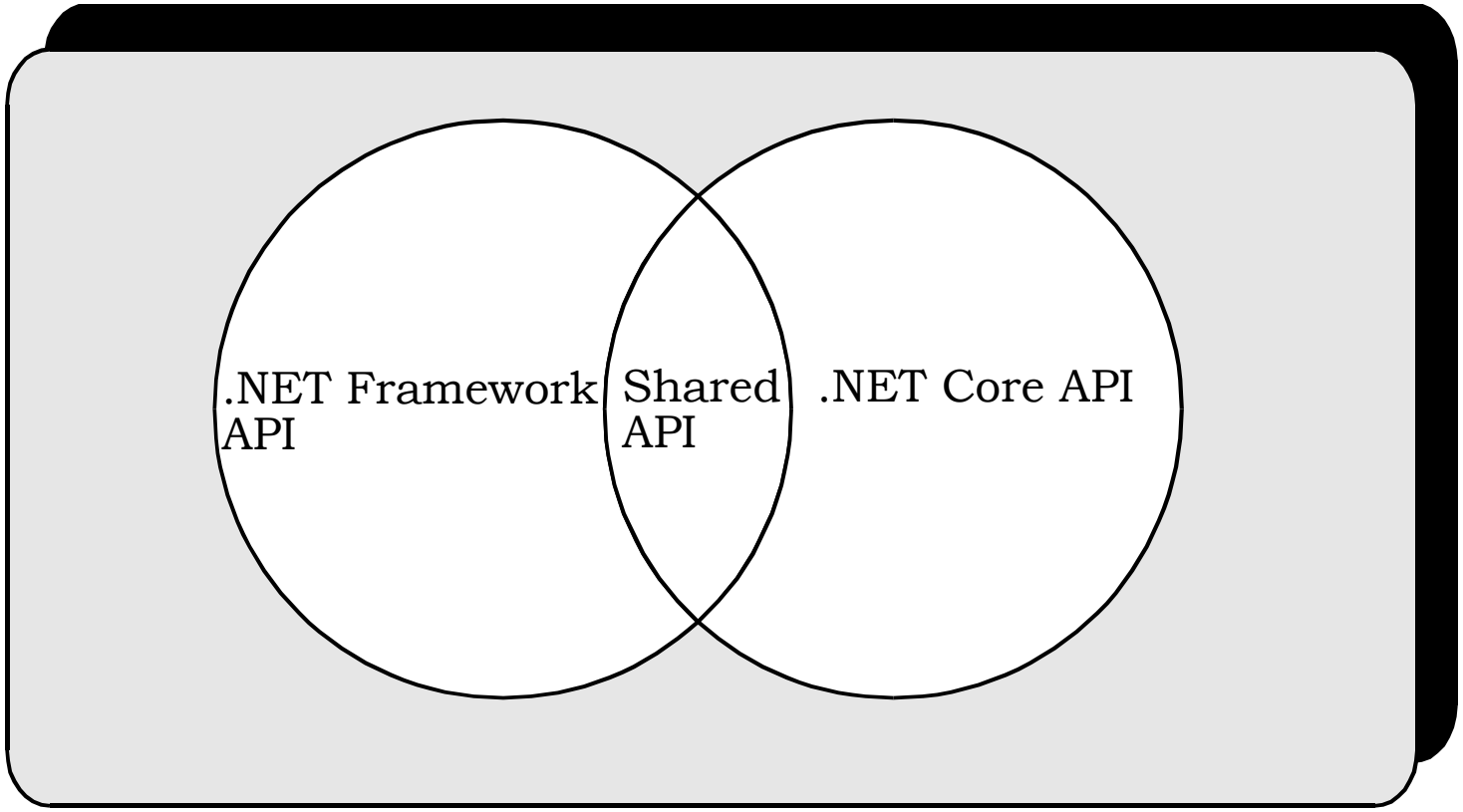


Web API Applications

- 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-PackageMicrosoft.AspNetCore.JsonPatch`
- Write in your controller

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

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

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:

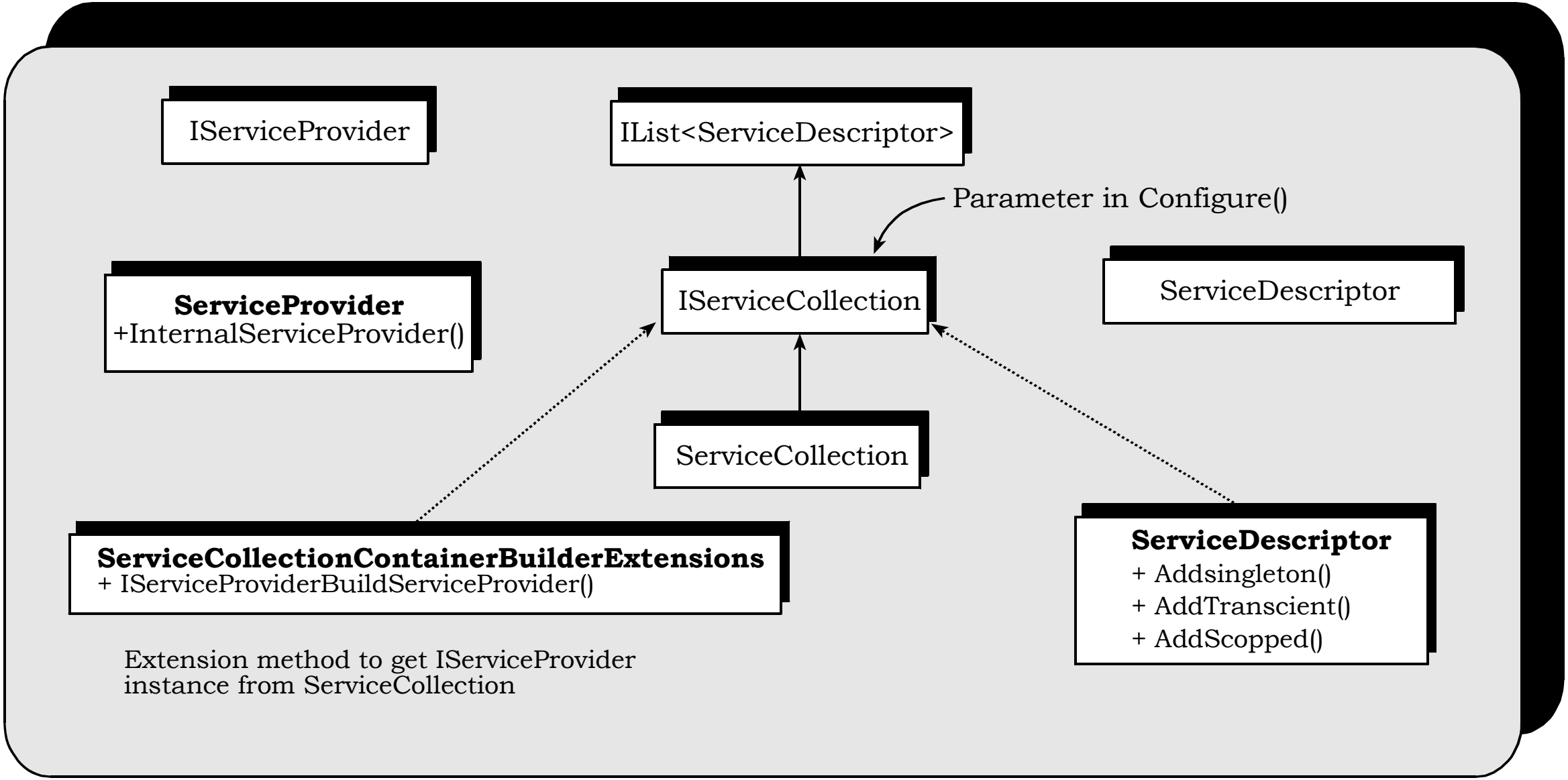
Interfaces

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();
}
```