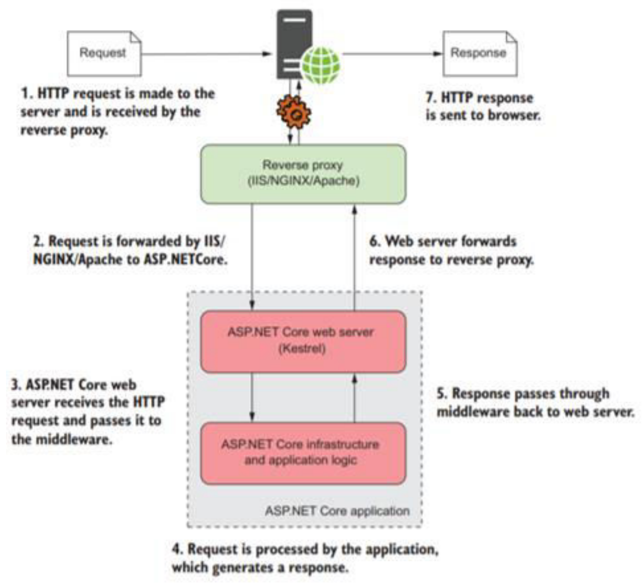
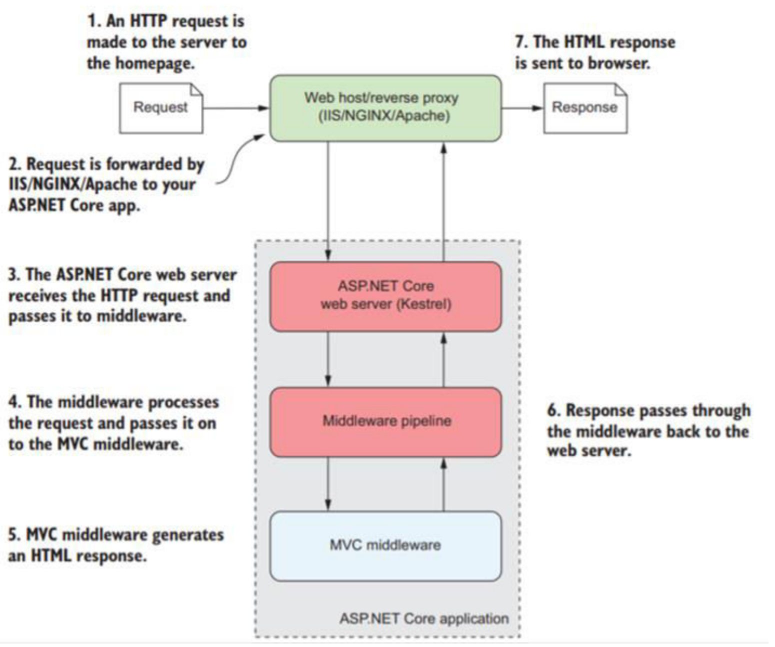
# Ch1 – Introduction

ASP.NET Core is a new web framework built with modern software architecture practices and modularization as its focus.
It's best used for new, "green-field" projects with few external dependencies.
Fetching a web page involves sending an HTTP request and receiving an HTTP response.
ASP.NET Core allows dynamically building responses to a given request.
ASP.NET Core can run on both .NET Framework and .NET Core.


Request
Response
1. HTTP request is made to the server and is received by the reverse proxy.
7. HTTP response is sent to browser.
Reverse proxy (IIS/NGINX/Apache)
2. Request is forwarded by IIS/NGINX/Apache to ASP.NETCore.
6. Web server forwards response to reverse proxy.
ASP.NET Core web server (Kestrel)
3. ASP.NET Core web server receives the HTTP request and passes it to the middleware.
5. Response passes through middleware back to web server.
ASP.NET Core infrastructure and application logic
ASP.NET Core application
4. Request is processed by the application, which generates a response.

# Ch2->5 – Middleware & Routing


1. An HTTP request is made to the server for the homepage.
Request
Web host/reverse proxy (IIS/NGINX/Apache)
7. The HTML response is sent to browser.
Response
2. Request is forwarded by IIS/NGINX/Apache to your ASP.NET Core app.
3. The ASP.NET Core web server receives the HTTP request and passes it to middleware.
ASP.NET Core web server (Kestrel)
4. The middleware processes the request and passes it on to the MVC middleware.
Middleware pipeline
6. Response passes through the middleware back to the web server.
5. MVC middleware generates an HTML response.
MVC middleware
ASP.NET Core application

Middleware: Pipeline of request handlers. Requests are run from the top down till one of them is accepted then the rest won't be called. Example of an MVC **Startup.cs**

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env){
    if (env.IsDevelopment()){
        app.UseDeveloperExceptionPage();
    }//Shows developer the error message in case of errors
    else{
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }//shows users a developer-made error page
    app.UseHttpsRedirection();//redirect HTTP to HTTPS
    app.UseStaticFiles();//handles any reuqested images, JS, and CSS
    app.UseCookiePolicy();
    app.UseMvc(routes =>{
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id}");
        routes.MapRoute(
            name: "second",
            template:
"{controller=contname}/{action=funcName}/{a}/{b}/{c}/{d}");
        routes.MapRoute(
            name: "thirdA",
            template: "{controller=cont2}/{attrib1}/{a2}");
    });//Add a new maproute for every page & their GET requests
```

Similar to middleware, routes will be matched from the top down till one matches. Example with default:

```
                                                        Cathces all
routes.MapRoute(    Means this attribute is optional    remaining attribs
    name: "thirdA",
    template: "{controller=cont2}/{attrib1}/{attrib2?}/{*others}");
    defaults: new {controller = "cname" , action = "actname"});
```

Example of actions inside a controller:

```
namespace Quiz.Controllers{
    public class StudentController : Controller{
        public string Index(string name, int age=1, string hobbies="na"){
            //Called by default when no action is specified
            return name+" is "+age+" with hobbies: "+hobbies;
        }
        public string noid(string name = "empty", int age=1, string
hobbies="none"){
            //Returns text on a blank page
            return name+" is " + age +" with hobbies: " + hobbies;
        }
        public IActionResult withid(int id = 0, string name = "default",
int age = 1, string hobbies = "none"){
            //Fills items into the viewbag to be used at the vew
            ViewBag.name = name;
            ViewBag.id = id;
            ViewBag.age = age;
            ViewBag.hobbies = hobbies;
            //Redirects to the named view
            return View("viewname");
            //NoteL if View() is empty, the view with the same name as
            //the action is used
        }
}
```

Instead of returning a view, we can return one of:

```
ViewResult(viewname) –Generates an HTML view.
RedirectResult(URL) –Sends a 302 HTTP redirect response to automatically
 send a user to a specified URL.
RedirectToAction(actionname) –Calls on the speccified action
NotFoundResult() –Sends a raw 404 HTTP status code as the response.
```

We can specify if this action is intended for post or get requests only:

```
[HttpPost]
public string Index(string name, int age=1, string hobbies="na"){…}
[HttpGet]
public string Index(string name, int age=1, string hobbies="na"){…}
```

# Ch6 – Binding Models

MVC uses C# classes (models) to send & receive data between views. When a model is set to be used on a page, upon sending a new request, the controller will attempt to fill this model from **Form values** with matching names, **Route values** from the URL, and **Querry Strings** (in that order). This check will happen for every attribute within the set class individually.

Example model:

```
namespace Exercise1.Models{
    public class Student{
        [DisplayName("First Name")][StringLength(maximumLength:20,MinimumLength =3)]
        public string Fname { get; set; }

        [DisplayName("Last Name")][StringLength(maximumLength: 20, MinimumLength = 3)]
        public string Lname { get; set; }

        [DataType("int")]
        public int Age { get; set; }

        [Required][Range(0,100)]
        public int Grade { get; set; }
    }
}
```

Note: if after all checks an attribute has no value, a default value will be used.

Some useful data bindings:
[CreditCard] [EmailAddress] [Phone] [Url]
[RegularExpression("Regex")] [Compare]
[Required(ErrorMessage="Field is Required")]

Example of a view that uses this model:

```
@model Exercise1.Models.Student
@{
    ViewData["Title"] = "Hello";
}
<center>
    <form asp-action="Show">
        <div class="form-group">
            <label asp-for="Fname"></label>
            <input asp-for="Fname" class="form-control" value="someDefualtValue" />
            <span asp-validation-for="Fname"></span>
        </div>
        <div class="form-group">
            <label asp-for="Lname"></label>
            <input asp-for="Lname" class="form-control" value="hello" />
        </div>
        <div class="form-group">
            <label asp-for="Age"></label>
            <input asp-for="Age" class="form-control" value="20" />
        </div>
        <div class="form-group">
            <label asp-for="Grade"></label>
            <input asp-for="Grade" class="form-control" value="0" />
        </div>
        <button class="btn btn-default" type="submit">Submit</button>
    </form>
</center>
```

This will call the "Show" action inside our Home Controller which is set to receive an object of type Student which looks like this:

```
public IActionResult Show(Student S){
    if (ModelState.IsValid)
        return View(S);
    return
RedirectToAction("Index");
```

Passing a model (S) with the return means that the target view (by default the one with the name as the action "Show") will receive this object like so:

```
@model Student
<center>
<h1> @Model.Fname @Model.Lname </h1>
</center>
```

# Ch7 – Using Razor views

**Razor**: (.cshtml) which is a mixture of C# and HTML which allows us to generate HTML dynamically

Instead of view models (binding models), we can also use ViewBag which uses C# dynamics objects and ViewData which is an associative array

Controller:

```
public IActionResult About(){
    ViewData["Message"] = "Hello.";
    ViewBag.test = "Hi.";
    return View();
}
```

In both cases, no parameters are needed in the return

About.cshtml:

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewBag.test</h2>
<h3>@ViewData["Message"]</h3>
```

Some loops we can use are:

```
@if (Model.someBoolvalue){}
else{}

@for(int i = 0; i < 8; i++){}

@while (true) { }

@foreach(int x in Model.someList) { }
```

**Note**: if we use @ to add in a string value, any HTML in that string won't follow through ex:

```
@{
    String x = "<strong> Hello </strong>";
}
<li> @x </li>
```

Output: **<li>&lt;strong&gt;Check oil&lt;/strong&gt;</li>**

Simple fix:

```
<li> @Html.Raw(x) </li>
```

**Layouts**: A razor file that is shared between multiple pages. Anything added to the layout is added to all pages that reference it. The only different between layouts and normal Razer files is that they must call @RenderBody() which his where the content of the referencing view will be inserted

Usually the <html> <head> and <body> tags, meta tags, shared CSS/JS file references, and environment tags are added here to limt redundancy

We can also use sections:

myLayout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewData["Title"] </title>
<script> … </script>
</head>
<body>
@RenderBody()
<hr/>
    @RenderSection("Extras", required: false)
</body>
    @RenderSection("Scripts", required: true)
</html>
```

Pages with this layout will have their @ViewData["Title"] as the title

This secton is not required and is only added if the view has it like:

```
@section Extras{
        ….
}
```

Otherwise it is ignored.

Reqired sections will cause an error if they aren't found in the view

**Partial Views**: Also razor files that are usually used by other views. The most common use is to be passed a model to render for ex:

someView.cshtml

```
@model List<item>
<ul>
@foreach (var cur in Model){
    @await Html.PartialAsync("myPartial", cur)
}
<ul>
```

myPartial.cshtml

```
@model item
<span class="fontStuff">
    @item.name : @item.price $
</span>
```

**_ViewStart.cshtml**: Holds common code that is run at the start of any full view in the project (doesn't apply to partial views or layout)

**_ViewImports.cshtml**: Holds directives inserted a the top of every full view like @using and @model statements

A file of these names can be placed in any folder and it will be applied to any view in the folder and any sub-folders