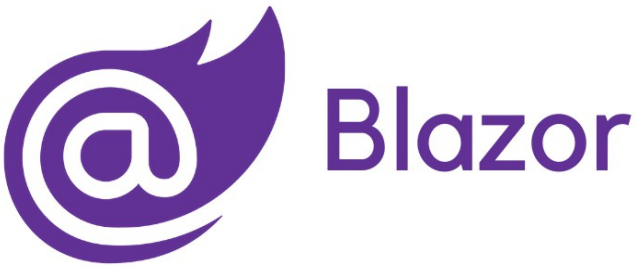


CSC530-B

Module 1

Beginning ASP.NET Application Development

1 Web Application Basics

1.1 Project Setup

Start Visual Studio 2022 and create a new project and solution using the information below. If you configure the project for HTTPS, your web server must have a certificate installed. A development certificate will be provided by .NET to enable HTTPS for local development and testing. If it is not already installed, you will be prompted to install it on the local IIS server on the first launch of the application. You can also install the developer certificate from command prompt by running **dotnet** with **dev-certs https** command. If will inform you if the certificate is already installed.

Install certificate from console

```
C:\WINDOWS\system32> dotnet dev-certs https
```

Project Information

```
Project Name     : Newrise
Project Template : Blazor Server App Empty
Location         : <repository>\src
Solution         : Module1
Options          : Configure for HTTPS
```

1.2 Application Host

Most applications require a basic set of components that provides standard application services such as logging, configuration and dependency injection. These components have to be built by the application developers themselves or provided by a third-party application framework. In ASP.NET, all the basic services are already available in the framework. The first step is to generate an application builder that will combine these standard services and components together with custom services and components to build an application host to host your web application.

For .NET version 5 and earlier, creation and configuration of the builder is split into 2 classes and files; **Program** and **Startup**. The application builder is created and setup to use the **Startup** class for configuration in the **Program** class. In .NET 6 and above creation and configuration of the builder has been simplified into a single class and method. Once the builder has been created and configured, call the **Build** method to generate the web application host and call its **Run** to start the application.

To create a host that has all required basic services and components to run a web application, create it by calling the **CreateBuilder** method on the **WebApplication** class. Then build and run your application host synchronously or asynchronously.

Building a web application host: Program.cs

```
class Program {
    static void Main(string[] args) {
        var builder = WebApplication.CreateBuilder(args);
        var app = builder.Build();
        app.Run();
    }
}
```

Asynchronous version:

```
class Program {
    static async Task Main(string[] args) {
        var builder = WebApplication.CreateBuilder(args);
        var app = builder.Build();
        await app.RunAsync();
    }
}
```

When you launch the application, Visual Studio will launch a web browser to access it. You can do this manually but you will need to know the network port the application is using. If you examine the console window opened by Visual Studio, you can see both the HTTP and HTTPS ports the application is listening on.

Example URLs to access the local application

```
https://localhost:7144
http://localhost:5020
```

1.3 Request & Response

A web application works on a request and response model. For every request, a web client or browser will expect a response. Currently our web application has no service or component to process web client requests so what you try to access the application you will get a HTTP 404 resource not found error. You can call **UseWelcomePage** on **app** to add a built-in middleware component that will construct a default web page to be send back as a response to any HTTP request. If you accessing the web application now, it will no longer return a HTTP error.

Adding a middleware component

```
static async Task Main(string[] args) {
    var builder = WebApplication.CreateBuilder(args);
    var app = builder.Build();
    app.UseWelcomePage();
    await app.RunAsync();
}
```

You can use **Run** and **Use** methods to map methods into the HTTP request processing pipeline. Following shows mapping an asynchronous anonymous method that return HTML content to show both the local and universal time on the server. A **HttpContext** object is passed to the method to provide the necessary functionality to communicate with the host to process the request and return the response.

Register method to return server time in HTML format

```
app.Run(async context => {
    context.Response.ContentType = "text/html";
    await context.Response.WriteAsync(
        "<fieldset><legend><strong>Server Time</strong></legend>" +
            $"<div>Local: {DateTime.Now}</div>" +
            $"<div>Universal: {DateTime.UtcNow}</div>" +
        "</fieldset>");
});
```

Use **Request** object from the context to access the request and use **Response** object to return a response. The **Run** method is terminal which means that it represents the final action in a HTTP pipeline. To chain multiple actions to run one after another, you must call **Use** method instead that accepts an additional **next** parameter representing a delegate to the next action to be executed in the chain. In the following example we break the previous action into two methods, one that returns local time and the other that returns universal time. **Use** will be terminal if there are no more actions added to the pipeline so there is no reason to explicitly use **Run** unless you want to guarantee that it will be a final action in the pipeline.

Register action to return local time only

```
app.Use(async (context, next) => {
    if (context.Request.Query["local"].Count > 0) {
        context.Response.ContentType = "text/html";
        await context.Response.WriteAsync(
            "<fieldset><legend><strong>Local Time</strong></legend>" +
            $"<div>{DateTime.Now}</div></fieldset>");
    } else await next();
});
```

Register action to return universal time only

```
app.Use(async (context, next) => {
    if (context.Request.Query["world"].Count > 0) {
        context.Response.ContentType = "text/html";
        await context.Response.WriteAsync(
            "<fieldset><legend><strong>Universal Time</strong></legend>" +
            $"<div>{DateTime.UtcNow}</div></fieldset>");
    } else await next();
});
```

A HTTP request can contain a query string. The above actions checks the query string using the **Query** collection on the **Request** object to see if certain a parameter exist before performing an action. You can now run the application and attach either *local* or *world* query parameter to the request URL. A query string is preceeded by question mark (?) symbol and can zero or more parameters separated by using an ampersand (&) symbol. A parameter can be assigned a value using equal (=) symbol. The query parameters for our above actions do not require values to be assigned. Their presence in the URL is good enough to activate the correct action.

Example URLs to retrieve local time or universal time

```
https://localhost:7144/?local
https://localhost:7144/?world
```

1.4 Middleware Components

To implement more complex or reusable actions, you should implement a middleware component class. The class should have a constructor to accept a delegate to execute the next action and a synchronous **Invoke** or asynchronous **InvokeAsync** method to accept the **HttpContext** object and process the request. Add a new project using the following information to implement our component.

Middleware component project

```
Project Name     : Newrise.Services
Project Template : Razor Library
```

Note than creating a *Razor Library* instead of a normal *Class Library* will also allow UI components to be included into the library, and not only code-based components. Add the following **ServerTimeMiddleware** class to the project. There are few differences between the class and the previous examples. This component can return the time for any time zone and the response will be in XML format instead of HTML.

Server time component: Newrise.Services\ServerTimeMiddleware.cs

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Newrise.Services {
    public class ServerTimeMiddleware {
        private readonly RequestDelegate _next;
        public ServerTimeMiddleware(RequestDelegate next) {
            _next = next;
        }
        public async Task InvokeAsync(HttpContext context) {
            var id = context.Request.Query["timezone"];
            if (id.Count > 0) {
                var zone = TimeZoneInfo.FindSystemTimeZoneById(id[0]);
                var time = TimeZoneInfo.ConvertTimeFromUtc(DateTime.UtcNow, zone);
                context.Response.ContentType = "text/xml";
                await context.Response.WriteAsync(
                    "<?xml version='1.0' encoding='utf-8'?>" +
                    $"<time zone='{zone.Id}'>{time}</time>");
            }
            else await _next(context);
        }
    }
}
```

You can now return back to your web application and reference the above library. Call the **UseMiddleware** generic method on the application host to add your component to the HTTP request processing pipeline.

Registering your middleware component: Newrise\Program.cs

```
app.UseMiddleware<ServerTimeMiddleware>();
```

URLs to test the component

```
https://localhost:7144/?timezone=Korea Standard Time
https://localhost:7144/?timezone=Pacific Standard Time
```

To simplify registration of middleware components you can implement an extension method to register the component. Extension method must be static method and the container class must also be static so we need to add another class for this method as shown in the following section.

Middleware registration extension method: Newrise.Services\Extensions.cs

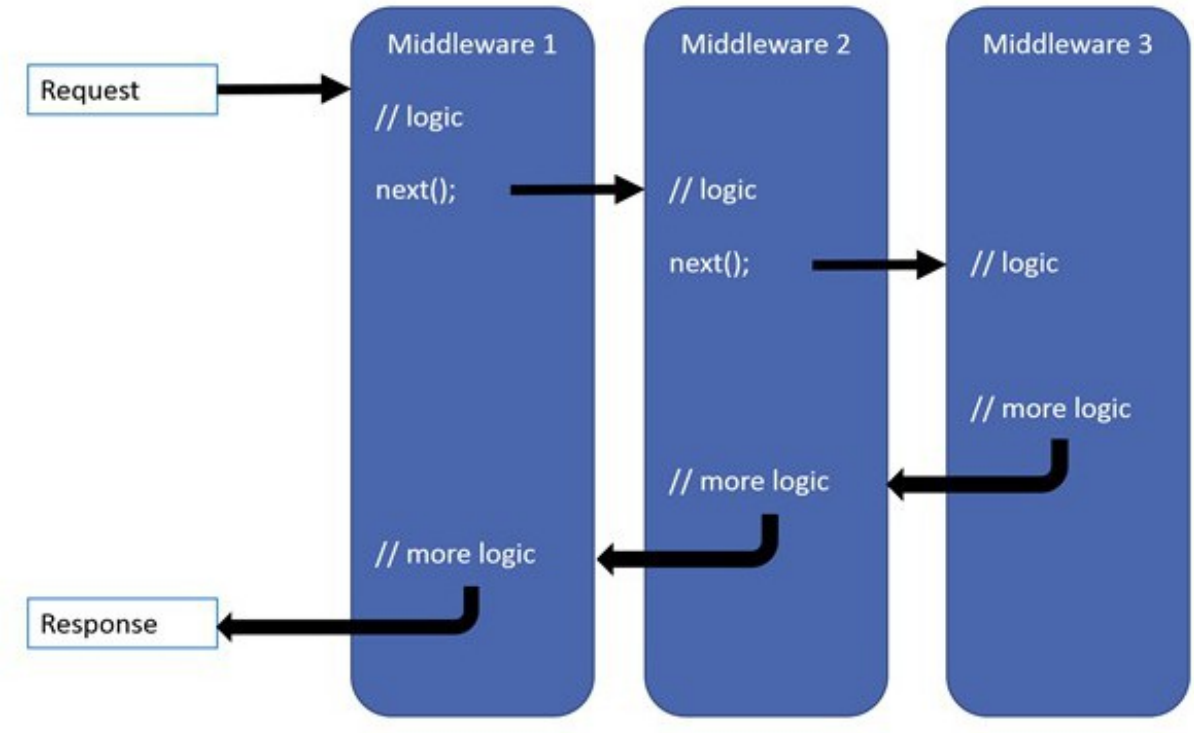
```
using Microsoft.AspNetCore.Builder;
namespace Newrise.Services {
    public static class Extensions {
        public static IApplicationBuilder UseServerTime(this IApplicationBuilder app) {
            app.UseMiddleware<ServerTimeMiddleware>();
            return app;
        }
    }
}
```

Simpler middleware registration syntax: Newrise\Program.cs

```
app.UseServerTime();
```

Understand that the execution of middleware components is not one after another but actually one inside the other. Thus the response of an inner component becomes the response of the outer component. Control goes back to the outer component once the inner component completes. The outer component can then have the ability to modify the response or replace the response of the next component in the pipeline. The order of registering middleware components is thus significant.

Middleware execution process in the HTTP pipeline



1.5 Essential Middleware

A numerical status code is returned in a HTTP response to indicate success or failure. A status code of 200 indicates a successful operation for normal requests. Status code in the 400 range indicates standard errors while in 500 range indicate server issues. At the moment, when runtime errors occur in a middleware, the web client will simply get a generic HTTP error response and a status code. There is not enough information for a developer to ascertain and resolve the issue. To help debug your application, you can use the **DeveloperExceptionPageMiddleware** by calling the following method. It would automatically generate a page to show the unhandled exception.

Using developer exception page middleware

```
app.UseDeveloperExceptionPage();
app.UseServerTime();
```

While this can assist debugging, it should be used for development and testing, and not for production. Allowing users to see the exception can allow hackers to locate bugs and weak points that they can then utilize to hack your application. The default environments are development, staging and production but custom environments can also be implemented if required. Use **Environment** property of the application host to access the current runtime environment. The following shows adding a component only when running under the development environment.

Enabling middleware only for development environment

```
if (app.Environment.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
}
```

HTTPS should always be enabled for Internet-based applications. This will guarantee full privacy between the web client and the server. Web browsers will now warn users if they tried to access sites that do not support HTTPS. If your site do support HTTPS you can call **UseHsts** method to add a middleware to inform the client to only talk to the server through HTTPS. You should not enable this in development as browsers will cache the setting and you have to clear the cache if you need to revert back to HTTP during development for faster performance and easier debugging. Normally the value is cached for up to 30 days.

Middleware to inform client to communicate securely

```
if (app.Environment.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
else {
    app.UseHsts();
}
```

Not every web browser supports HTTP Strict Transfer Security (HSTS), so you should also call **UseHttpsRedirection** to add a middleware to redirect the client to HTTPS if a HTTP request is received.

Middleware to redirect HTTP to HTTPS

```
app.useHsts();
app.UseHttpsRedirection();
```

A web application does not completely run on the web server. Additional resource and code files have to be downloaded to the web browser for local access and execution. Call **UseStaticFiles** method to add a middleware to enable a client to download static files in a specific directory. This directory can be customized by passing in an optional **StaticFileOptions** instance that contains the directory name but the default directory will be **wwwroot**. Notice this directory is already created for you if you selected any ASP.NET application project template.

Middleware to enable static file access

```
app.UseStaticFiles();
```

Copy the following files provided by the instructor into the default web directory. You should now be able to access these files from your web application.

Files copied into default web directory: \wwwroot\images

```
newrise-icon.png
newrise-logo.png
```

Accessing static files

```
https://localhost:7144/images/newrise-icon.png
https://localhost:7144/images/newrise-logo.png
```

1.6 Routing & Endpoints

Using the static files middleware is good enough to publish a static web site. However you would need to write code to implement a dynamic web site. We have seen how to use **Run** and **Use** methods to execute code, and implement a middleware component but using these techniques alone to implement a complete web site is too complicated and time-consuming. To simplify this process, you can use routing to segregate a site into virtual pages or actions, thus dividing its complexity. Call **UseRouting** method to enable routing. You can then map code to each route. One way to map code to routes is to call **UseEndpoints** method from where you can define routes and map each one to code that can then be activated through a URL that matches the route.

Enable routing middleware

```
app.UseRouting();
```

The following code simulates a web site with three pages; a default home page, about page and contact page. You can now run the web application and activated the code to generate each page by specifying the correct route in the URL. However in the end, you should always use a full web application framework that provides the easiest way to build a full-fledge web application with dynamic web pages.

Enable routing middleware

```
app.UseEndpoints(endpoints => {
    endpoints.MapGet("/", async context =>
        await context.Response.WriteAsync("<h1>Home page</h1>"));

    endpoints.MapGet("/about", async context =>
        await context.Response.WriteAsync("<h1>About page</h1>"));

    endpoints.MapGet("/contact", async context =>
        await context.Response.WriteAsync("<h1>Contact page</h1>"));
});
```

Example URLs to endpoint routes

```
https://localhost:7144/
https://localhost:7144/about
https://localhost:7144/contact
```

1.7 Web Application Frameworks

Currently there are 3 major web application frameworks that you can use to build a full ASP.NET Core web application; MVC, Razor Pages and Blazor. For developers that are accustomed to desktop application development, MVC is the appropriate choice as HTTP requests will be routed to methods in special classes called as **Controllers** to be processed and the method chooses the appropriate **View** to present the result. There is a clear separation of business logic and presentation. To build an application host to support MVC, call a **AddControllersAndViews** method on **Services** property before building the host.

Add MVC support to web application: Program.cs

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersAndView();
var app = builder.Build();
```

You can then call **MapControllerRoute** on the application host to map routes to the controller classes placed into the root **Controllers** folder. Since pattern matching can be used to map a URL to a controller, you can use a single route mapping to map to multiple controllers and action methods in the controller. To demonstrate MVC, add 3 folders to your project; **Models**, **Controllers** and **Views**. Add the following mapping where the default controller is **Home** and the default action is **Index** and the action method has an optional **id** parameter. The name of the mapping is not important and only used as a reference to access back that specific route mapping.

MVC route mapping

```
//  app.UseServerTime();
//  app.UseWelcomePage();

app.MapControllerRoute("default",
    "{controller=Home}/{action=Index}/{id?}");
```

Models in MVC are objects used to store input data or output results. We will first add a **ServerTimeModel** class to store the time information to be presented back to the user. Models that are only used for presentation are called view models.

Adding a model class: Models\ServerTimeModel.cs

```
namespace Newrise.Models {
    public class ServerTimeModel {
        public string Id { get; set; }
        public DateTime Time { get; set; }
        public ServerTimeModel(string id, DateTime time) {
            Id = id; Time = time;
        }
    }
}
```

You can now add the controller class in the **Controllers** folder. A controller class must extend directly or indirectly from the base **Controller** class. All action methods must return an object that implements **IActionResult** interface whose purpose is to return results back to the user.

A default controller with default action: Controllers\HomeController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace Newrise.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            return View();
        }
    }
}
```

The following is a list of all types and their methods that a controller can use to return the results back to the user. A **View** is the default action result but you can choose to return something rather than a view.

List of ActionResult types and methods

ContentResult	Content()	*return text-based content to client*
FileContentResult	File()	*download text/binary data as file to client*
JsonResult	Json()	*serialize object to client in JSON format*
JavaScriptResult	JavaScript()	*return content to client as Javascript*
RedirectResult	Redirect()	*redirect client to a different site*
RedirecttoRouteResult	RedirectToAction()	*redirect using action and controller name*
RedirectToRouteResult	RedirectToRoute()	*redirect using route string*
PartialViewResult	PartialView()	*updates existing view on the client*
ViewResult	View()	*renders a new view to the client*

The following implementation of the **Index** action method localizes the server time to a specific time zone based on the optional **id** parameter. The results will be stored in a **ServerTimeModel** object and presented back to the client in **JSON** format. If there is no **id** parameter in the request, an error message will be presented back in **HTML** format instead, although in practice you should return success and error results in the same format.

Processing and returning results in action methods

```
public IActionResult Index(string? id) {
    if (id != null) {
        id = id.Replace("_"," ");
        var zone = TimeZoneInfo.FindSystemTimeZoneById(id);
        var time = TimeZoneInfo.ConvertTimeFromUtc(DateTime.UtcNow, zone);
        var data = new ServerTimeModel(id, time);
        return Json(data);
    }
    return Content(
        "<h3>No time zone specified</h3>",
        "text/html");
}
```

URLs to test controller and action

```
https://localhost:7144
https://localhost:7144/?id=Korea_Standard_Time
https://localhost:7144/?id=Pacific_Standard_Time
https://localhost:7144/Home/Index/Korea_Standard_Time
https://localhost:7144/Home/Index/Pacific_Standard_Time
```

Complex results can be displayed to the client in a web page. This is the **View** part of MVC. Change the action result method from **Json** to **View**. You can specify a specific view to use or let ASP.NET MVC automatically determine the name and location of the view based on the controller and method.

Return default view for action method

```
public IActionResult Index(string? id) {
    if (id != null) {
        id = id.Replace("_"," ");
        var zone = TimeZoneInfo.FindSystemTimeZoneById(id);
        var time = TimeZoneInfo.ConvertTimeFromUtc(DateTime.UtcNow, zone);
        var data = new ServerTimeModel(id, time);
        return View(data);
    }   return Content("<h3>No time zone specified</h3>","text/html");
}
```

Under **Views** folder, add a subfolder named **Home** and create an empty Razor View in it named **Index.cshtml**. The subfolder has to follows the controller name and view file follows the action method name. Using this naming convention, ASP.NET Core can find the default view based on the controller and action.

Implementing a view: Views\Home\Index.cshtml

```
@model Newrise.Models.ServerTimeModel
<html>
<head><title>Server Time</title></head>
<body>
    <fieldset>
        <legend><strong>Server Time</strong></legend>
        <div>Time Zone: <i>@Model.Id</i></div>
        <div>Time: <i>@Model.Time</i></div>
    </fieldset>
</body>
</html>
```

Razor is a templating engine that can generate web page content from a combination of static content like HTML, CSS and text and dynamic content generated by code. In Razor, use **@** to switch between static content and code. If a view model is passed to the view from a controller, use **@model** directive to specify the type of model and use the **Model** property to access the view model object. Test the web application again with the previous URLs and you can see the results presented in the view.

Razor Pages may be much more preferable for the developers used to implementing web applications with ASP.NET Web Forms or ASP classic. There are no separation of business logic and presentation. Input, processing and output is typically done in the one page. To reduce complexity, complex processing are implemented as services to be injected for use into the page. Call **AddRazorPages** to build an application host to support this web framework. Once the application host is built, call **MapRazorPages** method to automatically map pages to routes. The actual route will be embedded into the page itself.

Add support for Razor Pages: Program.cs

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();
var app = builder.Build();
```

Enable automatic page route mapping

```
app.MapRazorPages();
```

Make sure that the application has a **Pages** folder. This is where you implement your Razor pages. Add an empty Razor page named **ServerTime**. Use **@page** directive to define the route for this page. Note that a view model is automatically added for you. Razor page view models must be extended from **PageModel** class.

Implementing a Razor page: Pages\ServerTimeModel.cshtml

```
@page "/time/{id?}"
@model Newrise.Pages.ServerTimeModel
<html>
<head><title>Server Time</title></head>
<body>
<fieldset>
    <legend><strong>Server Time</strong></legend>
    <div>Time Zone: <i>@Model.Id</i></div>
    <div>Time: <i>@Model.Time</i></div>
</fieldset>
</body>
</html>
```

You can now locate and implement the page model class. You can add properties and methods that can be accessed from the page through the **Model** property. Depending on the type of request, you can add a corresponding method to process the request. Add a **OnGet** method for HTTP GET request, **OnPost** method for HTTP POST request and so on. A POST request is usually used for sending data from a form in a web page for processing.

Implementing the Page model class

```
namespace Newrise.Pages
{
    public class ServerTimeModel : PageModel
    {
        public string? Id { get; set;  } = "Universal";
        public DateTime Time { get; set; } = DateTime.UtcNow;

        public void OnGet(string? id) {
            if (id != null) {
                id = id.Replace("_", " ");
                var zone = TimeZoneInfo.FindSystemTimeZoneById(id);
                var time = TimeZoneInfo.ConvertTimeFromUtc(DateTime.UtcNow, zone);
                Id = id; Time = time;
            }
        }
    }
}
```

Note that Razor pages themselves are components that can be distributed in a library. If you move the above **ServerTime** page to **Newrise.Services** library in the **Pages** folder under **Areas\MyFeature**, and rebuild the solution, the web application will still work without issue.

The newest web framework is **Blazor**. This allows you to implement web applications that has the same level of user interactivity as like desktop applications. This provides the features that are available on Javascript frameworks like Angular, React and Vue but you can implement your application completely in C#. Your C# code can interact with Javascript code as well if you wish. Currently there are three ways to implement a Blazor application; **Blazor Server**, **Blazor Webassembly** and **Auto** which basically combines both. The following modules of this course will concentrate on using Blazor but a number of topics can still be applied to all web frameworks including MVC and Razor pages.

2 Blazor Server Basics

2.1 Blazor Server Setup

A Blazor application is a single-page application where the content is constructed from a set of components. Razor is used to build the hosting page and the components so **Razor Pages** is required for Blazor applications. Call the **AddRazorComponents** and **AddInteractiveServerSideComponents** methods to build an application host that has support for Razor pages and Blazor components. Following that you can add the standard set of middleware components as shown below.

Add support for Razor pages and Blazor: Program.cs

```
static class Program {
    static void Main(string[] args) {
        var builder = WebApplication.CreateBuilder(args);
        builder.Services.AddRazorComponents().AddInteractiveServerComponents();
        var app = builder.Build();
                :
}
```

Middleware components to add

```
if (!app.Environment.IsDevelopment()) {
    app.UseExceptionHandler("/Error");   // a Razor page to handle and display errors to users
    app.UseHsts();                       // cached HTTPS setting on browser
}

app.UseHttpsRedirection();   //  always redirect to HTTPS
app.UseStaticFiles();        //  allows static files to be accessible
app.UseAntiforgery();        // to protect against CSRF attacks
```

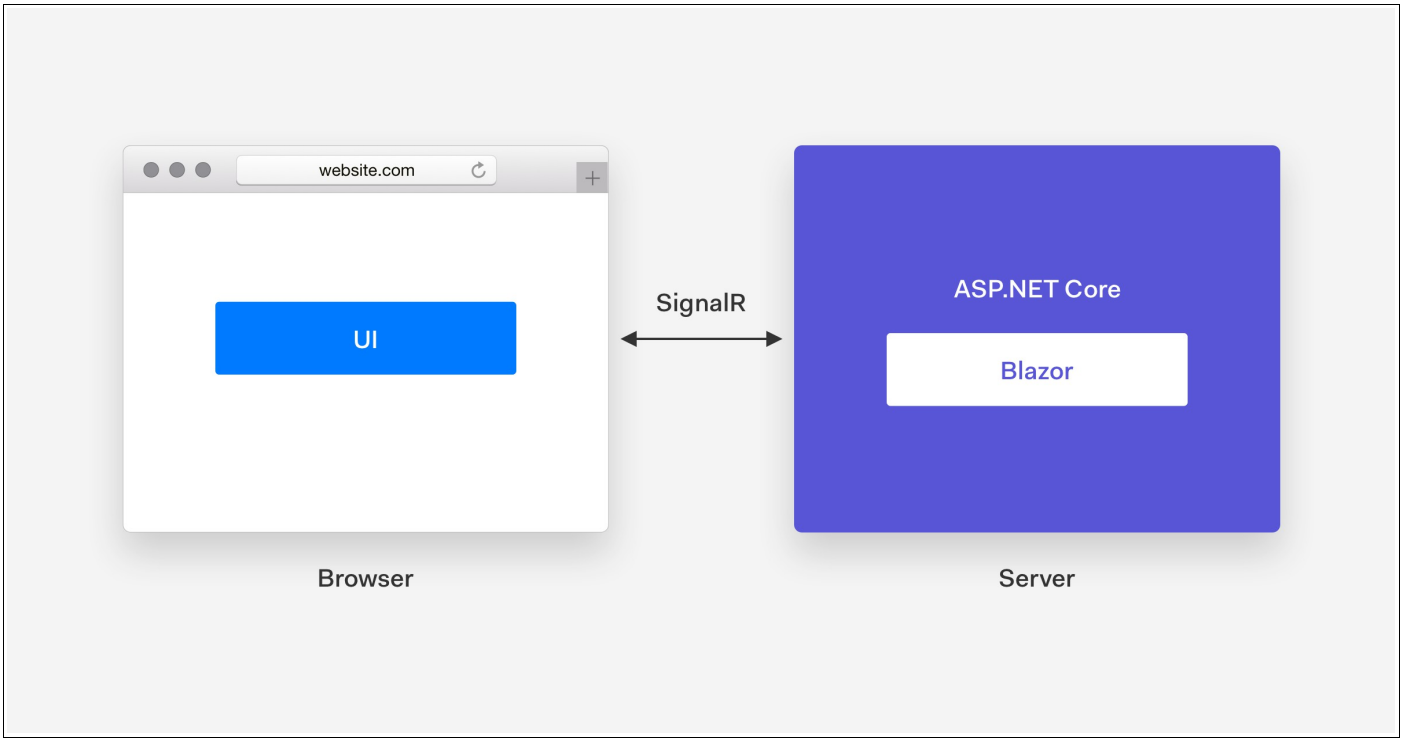
You can then setup routing to Razor pages by calling **MapRazorComponents** method and specify the hosting page class name. In the following code the class name is **App** which maps to the **App.razor** page.

Setup routing and enable SignalR

```
app.MapRazorComponents<App>().AddInteractiveServerRenderMode();
app.Run();
```

In a single-page server-side Blazor application, once the hosting page is fetched from an initial request, is not fetched again. Instead the web client will open a Websocket network connection back to the server to communicate with Blazor using a messaging system called **SignalR**. When Blazor components are updated by code running on the server, the changes are sent back to the web client to update the local UI. This keeps the client's UI state synchronized with the server. This is called as Interactive Server Rendering mode in Blazor.

Blazor server application model



2.2 Hosting Page

The hosting page is basically a Razor page named **App.razor** in the project. The page also embeds a component to support insertion and updating of content in the **HEAD** section of a web page and a custom component named **Routes** to setup the routing for Blazor components.

Hosting page: Components/App.razor

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="/" />
    <link rel="stylesheet" href="bootstrap/bootstrap.min.css" />
    <link rel="stylesheet" href="app.css" />
    <link rel="stylesheet" href="Newrise.styles.css" />
    <link rel="icon" type="image/png" href="favicon.png" />
    <HeadOutlet />
</head>
<body>
    <Routes />
    <script src="_framework/blazor.web.js"></script>
</body>
</html>
```

2.3 Blazor Routing

The front-end of a Blazor application is constructed from a set of Razor components. While ASP.NET provide routing to pages, you need a **Router** component for routing to components that are to be presented as separate pages even though technically they are hosted within a single web page. The default root component is **App**. To support the concept of pages, a root component will implement routing rather than presenting actual application content using **Router** component with the **AppAssembly** attribute set to the current application assembly. This allows the router to scan all the Razor components looking for the **@page** directive to map routes to those components. If a component matches a route, a **RouteView** component is used to present a matching component and sets the default layout for that component if the component does not set its own layout in the **Found** section. In the **NotFound** section, you can present content or components to indicate that the router has failed to locate a matching component.

Basic Blazor component routing: Routes.razor

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)" />
</Found>
</Router>
```

A Razor component can support more than one route with multiple **@page** directives. Internally the compiler will generate a .NET **Route** attribute for each directive so that is can be located by a router from type information. You can also add this attribute to the component using the **@attribute** directive.

Attaching Route attributes to a component

```
@page "/"
@page "/home"
@attribute [Route(Routes.Home)]
```

Reusable Razor components can be implemented in separate libraries. You can inform the **Router** to scan those libraries using the **AdditionalAssemblies** attribute to pass a array containing references to each of those assemblies. You need to refer to a type from the assembly to obtain the assembly reference.

Scanning external libraries for Razor component pages

```
<Router
    AppAssembly="@typeof(App).Assembly"
    AdditionalAssemblies="new[] {
        typeof(Lib1.Typename).Assembly,
        typeof(Lib2.Typename).Assembly,
        typeof(Lib3.Typename).Assembly
     }">
            :
</Router>
```

Route parameters can be binded to component properties by marking the properties with a .NET **Parameter** attribute. Use a **@code** directive to add members in a Razor component. Override **OnParametersSet** or **OnParametersSetAsync** method that is called when parameters are changed.

Example of a Razor component page: Pages\ServerTime.razor

```
@page "/time/{id?}"

<fieldset>
    <legend><strong>Server Time</strong></legend>
    <div>Time Zone: <i>@Id</i></div>
    <div>Time: <i>@Time</i></div>
</fieldset>

@code {
    [Parameter]
    public string? Id { get; set; }
    public DateTime Time { get; set; }

    protected override void OnParametersSet() {
        try {
            Id = Id.Replace("_", " ");
            var zone = TimeZoneInfo.FindSystemTimeZoneById(Id);
            Time = TimeZoneInfo.ConvertTimeFromUtc(DateTime.UtcNow, zone);
        }
        catch (Exception) {
            Id = "Universal";
            Time = DateTime.UtcNow;
        }
    }
}
```

2.4 Layout Component

You can have multiple layout components in an application. Each component page can use the **@layout** directive to assign a layout. The content of the component will then be inserted into location of the **@Body** directive in the layout. You can set the default layout to use in **RouteView** or **LayoutView** components. **LayoutView** is used when you need a layout for content that is not within a component page. A layout requires to **@inherit** from a specific **LayoutComponentBase** class instead of a default class for Razor components.

Selecting layout in RouteView and LayoutView: App.razor

```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Example layout component: MainLayout.razor

```
@inherits LayoutComponentBase
<header>
    <div>
        <img src="images/newrise-logo.png" />
    </div>
    <br />
</header>
<main>@Body</main>
<footer>
    <br />
    <div>
        <center>
            <small>Copyright &copy; Symbolicon Systems 2023</small>
        </center>
    </div>
</footer>
```

Even though a Razor component is not a web page, you can still change the title of the hosting page and insert content into the head section of the web page using the **PageTitle** and **HeadContent** components.

Selecting layout in RouteView and LayoutView: Routes.razor

```
<Router AppAssembly="@typeof(App).Assembly">
                :
    <NotFound>
        <PageTitle>Page Not Found</PageTitle>
        <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

2.5 Global Imports

Although you can use **@using** directives to import namespaces in Razor components, you can use the special **_Imports** component to import all the common namespaces so you do not need to import per component or specify the full type name. Create a **Layout** folder and move **MainLayout** into the folder. When you rebuild, this class will now be known as **Layout.MainLayout** instead in the project. You will see errors from the **App** component that refers to the layout. Adding **Newrise.Layout** to **_Imports** will resolve the problem.

Global namespace imports for Razor components: _Imports.razor

```
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using Newrise
@using Newrise.Layout
```

2.6 Component Style

While you can assign stylesheets in the hosting page to apply styles across the entire application, each Razor component or component page can have their own personal stylesheet. You can use a HTML **style** element to embed a stylesheet within the Razor component file itself but it is more cleaner to create a separate stylesheet instead. For a component stylesheet, make sure the stylesheet follows the component name. The following shows the stylesheet for the **ServerTime** component.

Component stylesheet: ServerTime.razor.css

```
fieldset {
    background-color: beige;
}

fieldset > legend {
    color: darkgray;
    font-style: italic;
}
```

The advantage of component styling is that it implements CSS isolation. Styles in one component will not conflict with styles for another component. The styles are only for that particular component. CSS isolation occurs at build time where component styles are processed and bundled into a single CSS. You must ensure the hosting page has a link to the bundled stylesheet. The bundled stylesheet will follow the application name so you just need to add the following link once to the hosting page.

Linking to CSS isolation stylesheet: App.razor

```
<head>
            :
    <link href="css/site.css" rel="stylesheet" />
    <link href="Newrise.styles.css" rel="stylesheet" />
            :
</head>
```

2.7 Error Page

When running in development environment, a detailed error page will be constructed automatically to show unhandled exceptions to allow the developer to fix the problem. Since this information should not be shown to users, you can create a separate error page to notify the user. If this is not implemented, Blazor will simply return a HTTP 500 internal server error code to the browser. There are also other ways to handle the errors in components so an error page is optional but still better than for the browser to show a HTTP 500 error message.

A simple error page: Pages/Error.razor

```
@page "/Error"
@using System.Diagnostics
<PageTitle>Error</PageTitle>
<h1 class="text-danger">Error.</h1>
<h2 class="text-danger">An error occurred while processing your request.</h2>
```

2.8 Launch Settings

If you wish to customize the HTTP and HTTPS port numbers or select the environment to use when launching the web application from Visual Studio, you will find a JSON file in the **Properties** folder named **launchSettings.json**. The launch profile to use can be selected on the toolbar on *Start Debugging* button right before the *Start Without Debugging* button. You can also select which web browser to use.

Application Launch Settings: Properties\launchSettings.json

```
{
    "iisSettings": {
        "iisExpress": {
            "applicationUrl": "http://localhost:21913",
            "sslPort": 44367
        }
    },
    "profiles": {
        "http": {
            "commandName": "Project",
            "dotnetRunMessages": true,
            "launchBrowser": true,
            "applicationUrl": "http://localhost:5174",
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        },
        "https": {
            "commandName": "Project",
            "dotnetRunMessages": true,
            "launchBrowser": true,
            "applicationUrl": "https://localhost:7144;http://localhost:5174",
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        },
        "IIS Express": {
            "commandName": "IISExpress",
            "launchBrowser": true,
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        }
    }
}
```