CitiusTech

Powering the Future of Healthcare

Asp.Net Core 6.0

Version 6.0

March 2023

This document is confidential and contains proprietary information, including trade secrets of CitiusTech. Neither the document nor any of the information contained in it may be reproduced or disclosed to any unauthorized person under any circumstances without the express written permission of CitiusTech.

Agenda

- **Introduction to Asp.Net Core**
- Getting Started
- Middleware Components
- Attribute Based Routing and Tag Helpers
- Validation using Annotation
- Dependency Injection
- Using Entity Framework Core 6.0
- Partial Views
- View Components
- Identity Framework

Introduction to Asp.Net Core (1/2)

- ASP.NET Core 6.0 is a new cross-platform framework for building modern cloud-based and on premise web applications
- ASP.NET Core 6.0 applications run on the .NET Core Framework as well as on the full .NET Framework
- We can develop and run ASP.NET Core 6.0 applications on Windows, Linux, and macOS
- The list includes
  - Ubuntu 14, 16
  - Linux Mint 17, 18
  - Debian 8
  - Fedora
  - CentOS 7.1 and Oracle 7.1
  - SUSE Enterprise Server 64 bits
  - OpenSuse 64 bits
  - macOS 10.11
  - macOS 10.12

Introduction to Asp.Net Core (2/2)

- Web Application development, can be done on Windows OS using Visual Studio or Visual Studio code and then the application can be deployed on any target system
- If required development can also be done on Linux and macOS using several system-specific source code editors
  - On Linux, we can use Visual Studio Code, VIM/VI, Sublime or Emacs
  - On macOS, we could use Visual Studio for Mac, Visual Studio Code or any other Mac-specific text editor
- The Visual Studio 2017 or Visual Studio Code developer environments would be the preferred choice though, since they provide everything necessary to be highly productive and to be able to debug and understand your code as well as navigate within it easily
- After building your application, you can use several web servers to run it
  - Apache
  - IIS
  - Kestrel self-host
  - Nginx

Asp.Net Core Development

- In Asp.Net Core 6.0 when we create a new asp.net core web application the MVC folder structure and configuration is not enabled by default
- The new project can be used for creating pages-based core web application which do not use a MVC structure
- For MVC based web development the MVC services has to be registered and MVC route has to be configured using middle ware concept explained next slides

Agenda

- Introduction to Asp.Net Core
- **Getting Started**
- Middleware Components
- Attribute Based Routing and Tag Helpers
- Validation using Annotation
- Dependency Injection
- Using Entity Framework Core 6.0
- Partial Views
- View Components
- Identity Framework

Getting Started (1/2)

- Every Asp.net core 6.0 web applications (Pages/MVC) is an executable console application. So, every web application is a self hosting entity
- The code below shows entry point function called Main() that does the hosting on kestrel server

```
public class Program
   {
      public static void Main(string[] args)
      {
         BuildWebHost(args).Run();
      }

      public static IWebHost BuildWebHost(string[] args) =>
         WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
   }
```

Getting Started (2/2)

- The Main function then hands over the control to Startup class to initialize the web application
- As a part of initialization the following functions are called:
  - The ConfigureServices function registers the services required. This registration is done using DI container
  - The Configure function creates a middleware pipeline as per requirement

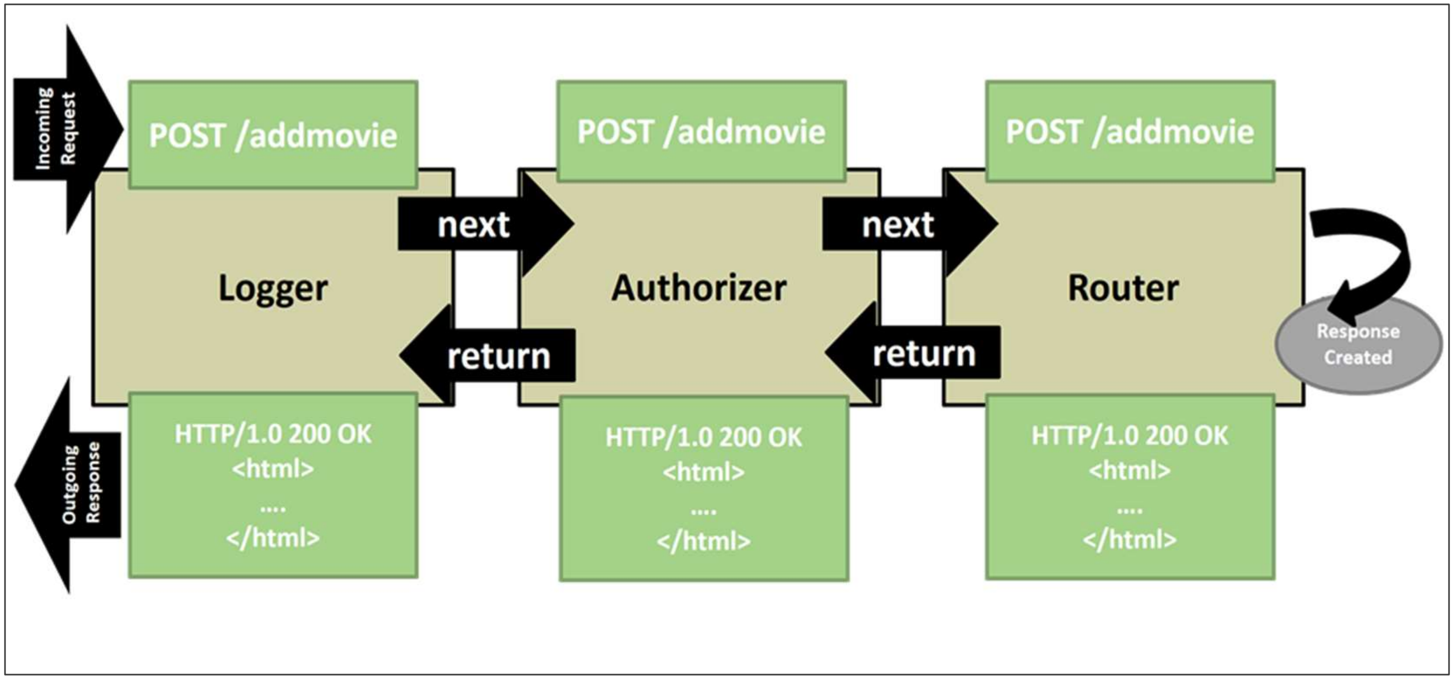
```
public void ConfigureServices(IServiceCollection services)
  {
            services.AddMvc();
  }
  public void Configure(IApplicationBuilder app,IHostingEnvironment env)
  {
    if (env.IsDevelopment())
      app.UseDeveloperExceptionPage();

            app.UseStaticFiles();
    app.Run(async (context) =>
    {
      await context.Response.WriteAsync("Hello World!");
    });
  }
```

Agenda

- Introduction to Asp.Net Core
- Getting Started
- **Middleware Components**
- Attribute Based Routing and Tag Helpers
- Validation using Annotation
- Dependency Injection
- Using Entity Framework Core 6.0
- Partial Views
- View Components
- Identity Framework

Middleware Components (1/3)

- Asp.net core has concept of linked middleware's. This concept is similar to filters and modules concept in the traditional MVC and Web forms development



https://odetocode.com/blogs/scott/archive/2016/11/22/asp-net-core-and-the-enterprise-part-3-middleware.aspx

Middleware Components (2/3)

```
//Inline Middleware…

//The middleware component below will return the same response whatever the url.


 app.Use(async (context, next) =>
{
     await context.Response.WriteAsync("Hello from a middleware!");
     await next.Invoke();
});

//Function as Middleware… [ Define the function and use ]
 async Task Process(HttpContext context, Func<Task> next)
 {
     await context.Response.WriteAsync("Hello from a middleware!!!");
     await next.Invoke();
 }

 app.Use(Process);
```

Middleware Components (3/3)

```
//Reusable Middleware Component [ Define a class and Use it ]
public class Middleware
    {
        private readonly RequestDelegate _next;
        public Middleware(RequestDelegate next)
        {
            this._next = next;
        }
        public async Task Invoke(HttpContext context)
        {
            await context.Response.WriteAsync("This is a middleware class!");
            await this._next.Invoke(context);
        }
    }
app.UseMiddleware<Middleware>();
```

Middleware Route Component (1/2)

- Enabling MVC Development - Default MVC Route

```
app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
//OR
 app.UseMvc(routes =>
            {
                routes.MapRoute(
                  name: "default",
                  template: "{controller}/{action}/{id?}",
                  defaults: new { controller = "Home", action = "Index" });
            });
//OR
 app.UseMvcWithDefaultRoute();
```

Middleware Route Component (2/2)

```
//Example 1
 app.UseMvc(routes =>
           {
               routes.MapRoute(
                   name: "default",
                   template: "{controller=Home}/{action=Index}/{id:int}");
           });
//Example 2
 app.UseMvc(routes =>
           {
               routes.MapRoute(
                 name: "default",
                 template: "{controller}/{action}/{id?}",
                 defaults: new { controller = "Home", action = "Index" },
                 constraints: new { id = new IntRouteConstraint() });
           });
//{page:range(1,100)}
//{isbn:regex(^\d{9}[\d|X]$)}
```

Custom Route Constraint (1/2)

```
public class EvenIntRouteConstraint : IRouteConstraint
  {
    public bool Match(
     HttpContext httpContext,
     IRouter route,
     string routeKey,
     RouteValueDictionary values,
     RouteDirection routeDirection)
    {
      if ((values.ContainsKey(routeKey) == false) || (values[routeKey] == null))
      {
        return false;
      }
      var value = values[routeKey].ToString();
      int intValue;
      if (int.TryParse(value, out intValue) == false)
      {
        return false;
      }
      return (intValue % 2) == 0;
    }
  }
```

Custom Route Constraint (2/2)

```
//To use Custom Route Constraint inline we can register it with services
 services.Configure<RouteOptions>(options =>
          {
             options.ConstraintMap.Add("evenint",
                                   typeof(EvenIntRouteConstraint));
          });
```

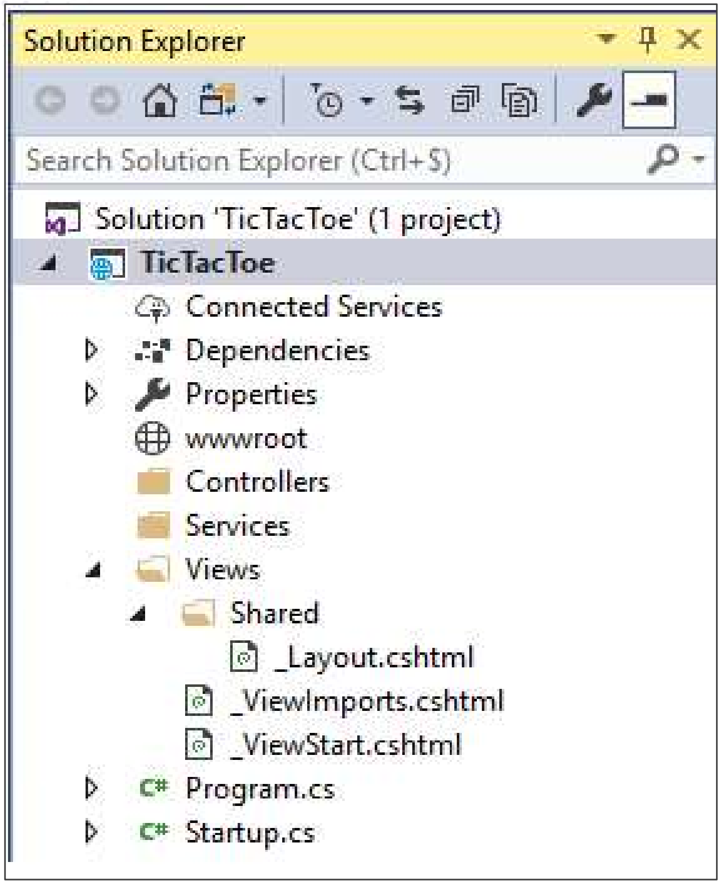
Agenda

- Introduction to Asp.Net Core
- Getting Started
- Middleware Components
- **Attribute Based Routing and Tag Helpers**
- Validation using Annotation
- Dependency Injection
- Using Entity Framework Core 6.0
- Partial Views
- View Components
- Identity Framework

Attribute Based Routing

```
//Example 1 Default Controller and default Action
 [Route("")]
 public class HomeController
{

        [Route("")]
        public IActionResult Index() { }
}
//Example 2: shows customizing the access url and inline constraint
[Route("Default/List")]
public IActionResult GetAll() { }

//Example 3 (with constraint)

[Route("Calculate({a:int},{b:int})")]
   public IActionResult Calculate(int a, int b) { }
```

Core MVC Project Structure



- Controllers: All Controller Classes
- Views: All CSHTML views
- Services: External Services
- _Layout: Common shared layout view
- _ViewStart: Decides the layout view to use
- _ViewImports: Common Namespaces (using)
- wwwroot: 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

Labs Hands On

Complete the Lab 1: Non MVC Web App and Lab 2: Getting Started with MVC before moving further

TagHelpers (1/4)

- 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
- There are many built-in Tag Helpers for common tasks, such as creating forms, links, loading assets etc. Tag Helpers are authored in C#
- The built-in tag helpers are all defined in Microsoft.AspNetCore.Mvc.TagHelpers core assembly
- Core framework also allows us to create custom tag helpers

TagHelpers (2/4)

- Let's take a look at some Tag Helper instructions in use:

```
//Generates a table that prints names and links for Editing and showing Details
<table>
  @foreach (var employee in Model.Employees)
  {
    <tr>
      <td>
          <a asp-controller="Home" asp-action="Details"
            asp-routeid="@employee.Id">Details</a>

          <a asp-controller="Home" asp-action="Edit"
                      asp-routeid="@employee.Id">Edit</a>
      </td>
      <td>@employee.Name</td>
    </tr>
  }
</table>
```

TagHelpers (3/4)

- Let's take a look at some Tag Helper instructions in use:

```
//Renders a Sign In form
<form asp-controller="Account" asp-action="Login" method="post" >
        <div asp-validation-summary="All" ></div>
        <div>
                <label asp-for="UserName" ></label>
                <input asp-for="UserName" />
                <span asp-validation-for="UserName" ></span>
        </div>
        <div>
                <label asp-for="Password"></label>
                <input asp-for="Password" />
              <span asp-validation-for="Password" ></span>
       </div>
        <div >
             <input type="submit"  value="Log in" />
        </div>
</form>
```

TagHelpers (4/4)

- To make use built in tag helpers add the following line in every chtml view or once in _ViewImports.cshtml

```
@addTagHelper "*, Microsoft.AspNetCore.Mvc.TagHelpers“
```

- If there are any user defined tag helper in the current project add the following line as well

```
@addTagHelper "*, <<your project name>>“
```

Agenda

- Introduction to Asp.Net Core
- Getting Started
- Middleware Components
- Attribute Based Routing and Tag Helpers
- **Validation using Annotation**
- Dependency Injection
- Using Entity Framework Core 6.0
- Partial Views
- View Components
- Identity Framework

Validation using Annotation (1/4)

- We can use the Data Annotations to add validation to our model
- The library contains attributes which can be applied to Model classes, such as:-
  - CompareAttribute – validate a value by comparing with another
  - RangeAttribute – validate if a value belongs in a given range
  - PhoneAttribute – validate if a value is a phone number
  - UrlAttribute – validate if a value is a URL
  - RequiredAttribute – validate if a value is supplied
  - CreditCardAttribute - validate if a value is a Credit Card number
  - EmailAddressAttribute – validate if a value is an Email Address
  - StringLengthAttribute – validate if a string has a specific length

Validation using Annotation (2/4)

- Applying attributes for specifying validation requirements

```
public class Product
  {
     public int Id { get; set; }

     [Required]
     [MaxLength(10)]
     public string Name { get; set; }

     [Required]
     public string Description { get; set; }

     [DisplayName("Price")]
     [RegularExpression(@"^\$?\d+(\.(\d{2}))?$")]
     public decimal UnitPrice { get; set; }
  }
```

Validation using Annotation (3/4)

- Applying attributes for specifying validation requirements

```
public class Doctor
{
  [Required]
  public int DoctorID { get; set; }
  [MinLength(3)]
  public string DoctorName { get; set; }
  [EmailAddress]
  public string DoctorEmail { get; set; }
  [Required]
  public string DoctorSpecialization { get; set; }
}
```

Validation using Annotation (4/4)

- The Controller action code which validates the data

```
[HttpPost]
  public ActionResult AddDoctor(Doctor doctor)
  {
    //Add a New Doctor from the User Input
    if(ModelState.IsValid)
    {
       doctorRepository.AddDoctor(doctor);
       return Redirect(“AllDoctors”);
     }
     else
     {
        ModelState.AddValidationError(“ErrorMessage”,”Something went wrong”);
        return View(“doctor”);
      }
  }
}
```

Remote Validation using Annotation

- The data validation annotations support in asp.net core also includes an annotation named “Remote”
- This is useful to perform a validation on server side , otherwise which we cannot accomplish using client-side script
- Validation is managed internally by making a AJAX call to a controller action on the server which return JSON result

Labs Hands On

- Complete the following labs:
  - Lab 3 MVC Web App - Client Side Validation
  - Lab 4 MVC Web App - Server Side Validation
- Assignment: Use the built in Tag Helpers to create the New Patient and Edit Patient CSHTML Views. You may create a separate views TagNew.cshtml and TagEdit.cshtml

Agenda

- Introduction to Asp.Net Core
- Getting Started
- Middleware Components
- Attribute Based Routing and Tag Helpers
- Validation using Annotation
- **Dependency Injection**
- Using Entity Framework Core 6.0
- Partial Views
- View Components
- Identity Framework

Dependency Injection Support (1/5)

- 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
- The built-in container is represented by IServiceProvider implementation
- The types (classes) managed by built-in IoC container are called services
- There are two types of services in ASP.NET Core:
  - Framework Services: Services which are a part of ASP.NET Core framework such as IApplicationBuilder, IHostingEnvironment, ILoggerFactory etc.
  - Application Services: The services (custom types or classes) which you as a programmer create for your application

Dependency Injection Support (2/5)

- Consider the following simple interface and service implementation

```
public interface ILog
    {
        void info(string str);
    }

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

Dependency Injection Support (3/5)

- Service Registration can be done one of the following ways in configureservices function of Startup class:

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

 // Transient
  services.Add(new ServiceDescriptor(typeof(ILog), typeof(MyConsoleLogger),
                                                        ServiceLifetime.Transient));
 // Scoped
   services.Add(new ServiceDescriptor(typeof(ILog), typeof(MyConsoleLogger),
                                                        ServiceLifetime.Scoped));
 }
```

Dependency Injection Support (4/5)

- An alternative way of registering the services

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddSingleton<ILog, MyConsoleLogger>();
            services.AddSingleton(typeof(ILog), typeof(MyConsoleLogger));

            services.AddTransient<ILog, MyConsoleLogger>();
            services.AddTransient(typeof(ILog), typeof(MyConsoleLogger));

            services.AddScoped<ILog, MyConsoleLogger>();
            services.AddScoped(typeof(ILog), typeof(MyConsoleLogger));
        }
```

Dependency Injection Support (5/5)

- 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:
  - Singleton: IoC container will create and share a single instance of a service throughout the application's lifetime
  - Transient: The IoC container will create a new instance of the specified service type every time you ask for it
  - Scoped: IoC container will create an instance of the specified service type once per request and will be shared in a single request

Agenda

- Introduction to Asp.Net Core
- Getting Started
- Middleware Components
- Attribute Based Routing and Tag Helpers
- Validation using Annotation
- Dependency Injection
- **Using Entity Framework Core 6.0**
- Partial Views
- View Components
- Identity Framework

Using EF Core 6.0

- Entity Framework has been added into dotnet core
- It is an ORM based framework and API to give a level of abstraction when connecting to databases
- Asp.Net Core web application can make use of EF Core 6.0 , to connect to databases
- Like full .net Entity Framework , here we need to create entity classes and ObjectContext classes.ObjectContext will act as connector to the database
- We can implement Code First or Database First approach

Labs Hands On

- Complete the Lab 5 MVC Web App - Using EF Core 6.0 & DI
- This lab shows how to use EF Core 6.0 and DI for custom application services

Agenda

- Introduction to Asp.Net Core
- Getting Started
- Middleware Components
- Attribute Based Routing and Tag Helpers
- Validation using Annotation
- Dependency Injection
- Using Entity Framework Core 6.0
- **Partial Views**
- View Components
- Identity Framework

Partial Views

- Asp.Net Core 6.0 supports Partial Views
- Partial View is a reusable UI piece
- It is to be included in other main Views and not directly accessed via URL
- It can be strongly-typed
- It is similar to UserControl concept of legacy web form development
- The partial view always gets the relevant data from its parent view

Agenda

- Introduction to Asp.Net Core
- Getting Started
- Middleware Components
- Attribute Based Routing and Tag Helpers
- Validation using Annotation
- Dependency Injection
- Using Entity Framework Core 6.0
- Partial Views
- **View Components**
- Identity Framework

View Component

- At times we might want to embed information in the main view which is not passed on to this view by the parent controller action
- View Components can be used in such cases
- View Components does not need to depend on parent view to pass the context information
- View Component can have connect to its own data source for relevant information
- This is similar to RenderAction feature available in full .net framework Asp.Net MVC development
- The code below renders the PriorityList View Component in the current view

```
@await Component.InvokeAsync("PriorityList",
                             new { maxPriority = 4, isDone = true })
```

- The above task can also been done using built in tag helper of View Components

```
<vc:priority-list max-priority="2" is-done="false"> </vc:priority-list>
```

Labs Hands On

- Complete the Lab 6 MVC Web App - Partial Views and View Component
- This Lab shows how to create and use Partial View and View Components

Agenda

- Introduction to Asp.Net Core
- Getting Started
- Middleware Components
- Attribute Based Routing and Tag Helpers
- Validation using Annotation
- Dependency Injection
- Using Entity Framework Core 6.0
- Partial Views
- View Components
- **Identity Framework**

Identity Framework (1/2)

- ASP.NET Core Identity framework is used to implement forms authentication
- There are many options to choose from for identifying your users including Windows Authentication and third-party identity providers like Google, Microsoft, Facebook, and GitHub etc.
- Following are the Core Players in usng Identity framework
  - User Class: This defines what the user profile information are we storing
  - UserStore<T> & IdentityDb<T> Classes: Defines the persistent store where user information is stored
  - SiginManager Class: Can sign in a user once we validate the password. the manager issues a cookie to the user's browser, and the browser will send this cookie on every subsequent request. It helps us identify that user
  - UserManager Class: This is front API class to create Users and Roles management
  - Identity Middleware: Reading the cookie sent by the SignInManager and identifying the user, this happens in the final piece of the framework, the Identity Middleware

Identity Framework (2/2)

- The Identify Framework for Core .Net can connect to third party providers like Azure AD, Google, Tweeter etc using OpenID and Oauth
- Following attributes are used to manage authorization
  - [Authorize]:
    - Ensures only valid and authenticated users have access to the resource
    - Can be applied at Controller or Action Level
  - [AllowAnonymous]: Useful to enable anonymous access to the resources

Labs Hands On

- Complete the Lab 7 Securing MVC Web App - Using Identify Framework
- This Lab shows how to implement authentication and authorization using Identity framework available with Asp.Net Core 6.0

Thank you!

Perspective from leaders across the healthcare industry

- On-Demand Webinars
- Reports & eBooks
- Workshops
- Success Stories
- Articles & Blogs
- News

Explore the Knowledge Hub >

CitiusTech

Powering the Future of Healthcare