ASP.NET CORE MVC

Apresentação do Formador

- Nome
- Funções
- Competências Profissionais

Apresentação dos Participantes

- Nome
- Empresa
- Expectativas sobre este curso

Conteúdo Programático

- Introdução ao ASP.Net Core
- ASP.NET Web Apps
- Dynamically render HTML with Razor
- Integration with data
- Secure web apps
  - Authorization & Authentication
- Scaffolding, forms, and validation
- Model View Controller (MVC)

Introdução ao ASP.Net Core

- Modern, scalable web apps with .NET and C#
- Use .NET and C# to create websites based on HTML5, CSS, and JavaScript that are secure, fast, and can scale to millions of users.
- Run on Windows, Linux, Mac, Android

Dynamically render HTML with Razor

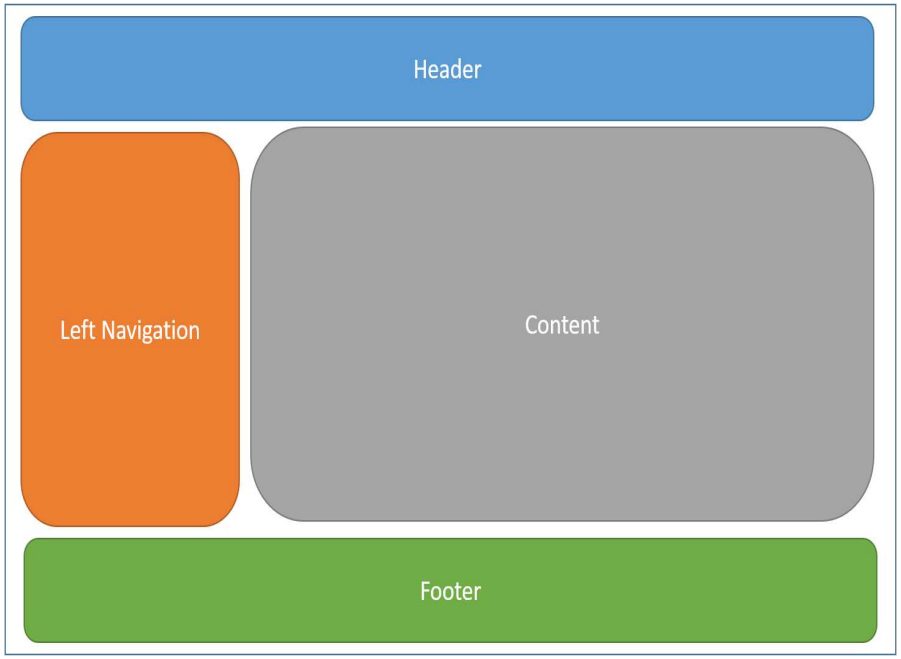
- **Layout in ASP.NET Core**
- **Razor syntax reference for ASP.NET Core**
  - The default Razor language is HTML. Rendering HTML from Razor markup is no different than rendering HTML from an HTML file. HTML markup in *.cshtml* Razor files is rendered by the server unchanged.
- **Razor class library**

Layout in ASP.NET Core

- Pages and views frequently share visual and programmatic elements. This article demonstrates how to:
- Use common layouts.
- Share directives.
- Run common code before rendering pages or views.
- This document discusses layouts for the two different approaches to ASP.NET Core MVC: Razor Pages and controllers with views. For this topic, the differences are minimal:
- Razor Pages are in the *Pages* folder.
- Controllers with views uses a *Views* folder for views.

Layout in ASP.NET Core

- Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page. The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.



Importing Shared Directives

- Views and pages can use Razor directives to import namespaces and use dependency injection. Directives shared by many views may be specified in a common _ViewImports.cshtml file. The _ViewImports file supports the following directives:

- @addTagHelper
- @removeTagHelper
- @tagHelperPrefix
- @using
- @model
- @inherits
- @inject

Running Code Before Each View

- Code that needs to run before each view or page should be placed in the *_ViewStart.cshtml* file. By convention, the *_ViewStart.cshtml* file is located in the *Pages* (or *Views*) folder. The statements listed in *_ViewStart.cshtml* are run before every full view (not layouts, and not partial views).
  Like ViewImports.cshtml, *_ViewStart.cshtml* is hierarchical. If a *_ViewStart.cshtml* file is defined in the view or pages folder, it will be run after the one defined in the root of the *Pages* (or *Views*) folder (if any).

Razor syntax

- Razor is a markup syntax for embedding server-based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a .cshtml file extension. Razor is also found in Razor components files (.razor).

- Rendering HTML
- The default Razor language is HTML. Rendering HTML from Razor markup is no different than rendering HTML from an HTML file. HTML markup in .cshtml Razor files is rendered by the server unchanged.

- Razor syntax
- Razor supports C# and uses the @ symbol to transition from HTML to C#. Razor evaluates C# expressions and renders them in the HTML output.

- When an @ symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup. Otherwise, it transitions into plain C#.

Razor syntax

- When an @ symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup. Otherwise, it transitions into plain C#.
- To escape an @ symbol in Razor markup, use a second @ symbol:
- CSHTML
  - @@Username
  - The code is rendered in HTML with a single @ symbol:
- HTML
  - @Username
  - HTML attributes and content containing email addresses don't treat the @ symbol as a transition character. The email addresses in the following example are untouched by Razor parsing:
- CSHTML
  - Support@contoso.com

Implicit Razor expressions

- Implicit Razor expressions start with @ followed by C# code:

- CSHTML
  - @DateTime.Now
  - @DateTime.IsLeapYear(2016)

With the exception of the C# await keyword, implicit expressions must not contain spaces. If the C# statement has a clear ending, spaces can be intermingled:

- CSHTML

  - @await DoSomething("hello", "world")

Implicit expressions cannot contain C# generics, as the characters inside the brackets (<>) are interpreted as an HTML tag. The following code is not valid:

- CSHTML
  - @GenericMethod<int>()

Explicit Razor expressions

- **Explicit Razor expressions consist of an @ symbol with balanced parenthesis. To render last week's time, the following Razor markup is used:**
  - Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))
  - @{ var joe = new Person("Joe", 33); } Age@(joe.Age)

Expression encoding

- **C# expressions that evaluate to a string are HTML encoded. C# expressions that evaluate to IHtmlContent are rendered directly through IHtmlContent.WriteTo. C# expressions that don't evaluate to IHtmlContent are converted to a string by ToString and encoded before they're rendered.**

- @("Hello World")

Razor code blocks

- Razor code blocks start with @ and are enclosed by {}. Unlike expressions, C# code inside code blocks isn't rendered. Code blocks and expressions in a view share the same scope and are defined in order:
  - @{ var quote = "The future depends on what you do today. - Mahatma Gandhi"; } @quote @{ quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr."; } @quote

Control structures

- Control structures are an extension of code blocks. All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures:

- Conditionals @if, else if, else, and @switch

- @if (value % 2 == 0) { The value was even. }

Directives

- Razor directives are represented by implicit expressions with reserved keywords following the @ symbol. A directive typically changes the way a view is parsed or enables different functionality.
- Understanding how Razor generates code for a view makes it easier to understand how directives work.
- @attribute, @code, @functions, etc..

ASP.NET Core built-in Tag Helpers

- Anchor Tag Helper
- Cache Tag Helper
- Component Tag Helper
- Distributed Cache Tag Helper
- Environment Tag Helper
- Form Tag Helper
- Form Action Tag Helper
- Image Tag Helper
- Input Tag Helper
- Label Tag Helper
- Link Tag Helper
- Partial Tag Helper
- Script Tag Helper
- Select Tag Helper
- Textarea Tag Helper
- Validation Message Tag Helper
- Validation Summary Tag Helper

Tag Helpers in forms in ASP.NET Core

- The Form Tag Helper:
- Generates the HTML <FORM> action attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the [ValidateAntiForgeryToken] attribute in the HTTP Post action method)
- Provides the asp-route-<Parameter Name> attribute, where <Parameter Name> is added to the route values. The routeValues parameters to Html.BeginForm and Html.BeginRouteForm provide similar functionality.
- Has an HTML Helper alternative Html.BeginForm and Html.BeginRouteForm

The Form Action Tag Helper

- Attribute Description
  - asp-controller The name of the controller.
  - asp-action The name of the action method.
  - asp-area The name of the area.
  - asp-page The name of the Razor page.
  - asp-page-handler The name of the Razor page handler.
  - asp-route The name of the route.
  - asp-route-{value} A single URL route value. For example, asp-route-id="1234".
  - asp-all-route-data All route values.
  - asp-fragment The URL fragment.

Integration with data

- The popular Entity Framework (EF) data access library lets you interact with databases using strongly typed objects.

- Most popular databases are supported, including SQLite, SQL Server, MySQL, PostgreSQL, DB2 and more, as well as non-relational stores such as MongoDB, Redis, and Azure Cosmos DB.

Session and state management in ASP.NET Core

- State management
  - State can be stored using several approaches. Each approach is described later in this topic.

- STATE MANAGEMENT
  - Storage approach Storage mechanism
  - Cookies HTTP cookies. May include data stored using server-side app code.
  - Session state HTTP cookies and server-side app code
  - TempData HTTP cookies or session state
  - Query strings HTTP query strings
  - Hidden fields HTTP form fields
  - HttpContext.Items Server-side app code
  - Cache Server-side app code

Demonstration

- Demo – Razor Pages

Exercício

Create this model:

```
public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

       [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
       public string Genre { get; set; }
       public decimal Price { get; set; }
    }
```

Exercício

Based on previous model: (in memory storage)

- Create a Razor Pages web app
- Add a model to a Razor Pages app (Movies)
- Update Razor pages
- Add search
- Add a new field
- Add validation

Session and state management in ASP.NET Core

- Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.

- EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.

- EF Core supports many database engines, see Database Providers for details.

Secure Web Apps

- Build secure web apps
- ASP.NET provides a built-in user database with support for multi-factor authentication and external authentication with Google, Twitter, and more.

- ASP.NET supports industry standard authentication protocols. Built-in features help protect your apps against cross-site scripting (XSS) and cross-site request forgery (CSRF).

ASP.NET Core Authentication

- Authentication is the process of determining a user's identity. Authorization is the process of determining whether a user has access to a resource. In ASP.NET Core, authentication is handled by the IAuthenticationService, which is used by authentication middleware. The authentication service uses registered authentication handlers to complete authentication-related actions. Examples of authentication-related actions include:
  - Authenticating a user.
  - Responding when an unauthenticated user tries to access a restricted resource.
- The registered authentication handlers and their configuration options are called "schemes".

- Authentication schemes are specified by registering authentication services in Startup.ConfigureServices:

- By calling a scheme-specific extension method after a call to services.AddAuthentication (such as AddJwtBearer or AddCookie, for example). These extension methods use AuthenticationBuilder.AddScheme to register schemes with appropriate settings.
- Less commonly, by calling AuthenticationBuilder.AddScheme directly.

ASP.NET Core Authentication

- The Authentication middleware is added in Startup.Configure by calling the UseAuthentication extension method on the app's IApplicationBuilder. Calling UseAuthentication registers the middleware which uses the previously registered authentication schemes. Call UseAuthentication before any middleware that depends on users being authenticated. When using endpoint routing, the call to UseAuthentication must go:

- After UseRouting, so that route information is available for authentication decisions.
- Before UseEndpoints, so that users are authenticated before accessing the endpoints.

ASP.NET Core Authorization

- Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create a document library, add documents, edit documents, and delete them. A non-administrative user working with the library is only authorized to read the documents.

- Authorization is orthogonal and independent from authentication. However, authorization requires an authentication mechanism. Authentication is the process of ascertaining who a user is. Authentication may create one or more identities for the current user.

ASP.NET Core Authentication

- The Authentication middleware is added in Startup.Configure by calling the UseAuthentication extension method on the app's IApplicationBuilder.
- Calling UseAuthentication registers the middleware which uses the previously registered authentication schemes.
- Call UseAuthentication before any middleware that depends on users being authenticated. When using endpoint routing, the call to UseAuthentication must go:

- After UseRouting, so that route information is available for authentication decisions.
- Before UseEndpoints, so that users are authenticated before accessing the endpoints.

Scaffolding, forms, and validation

- Quickly scaffold user interfaces to interact with your data model, including query and update.
- Dynamically generate HTML forms based on your strongly typed data model. Declaratively define validation rules, using C# attributes, which are applied on the client and server.

ASP.NET Core MVC

- The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns.
- Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

ASP.NET Core MVC Views

- Views help to establish separation of concerns within an MVC app by separating the user interface markup from other parts of the app. Following SoC design makes your app modular, which provides several benefits:
  - The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.
  - The parts of the app are loosely coupled. You can build and update the app's views separately from the business logic and data access components. You can modify the views of the app without necessarily having to update other parts of the app.
  - It's easier to test the user interface parts of the app because the views are separate units.
  - Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

ASP.NET Core MVC Partial Views

- A partial view is a Razor markup file (*.cshtml*) that renders HTML output *within* another markup file's rendered output.
- The term *partial view* is used when developing either an MVC app, where markup files are called *views*, or a Razor Pages app, where markup files are called *pages*. This topic generically refers to MVC views and Razor Pages pages as *markup files*.

- Partial views are an effective way to:
  - Break up large markup files into smaller components.
  - In a large, complex markup file composed of several logical pieces, there's an advantage to working with each piece isolated into a partial view. The code in the markup file is manageable because the markup only contains the overall page structure and references to partial views.
  - Reduce the duplication of common markup content across markup files.
  - When the same markup elements are used across markup files, a partial view removes the duplication of markup content into one partial view file. When the markup is changed in the partial view, it updates the rendered output of the markup files that use the partial view.

ASP.NET Core MVC Controllers

- A controller is used to define and group a set of actions. An action (or action method) is a method on a controller which handles requests. Controllers logically group similar actions together. This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively. Requests are mapped to actions through routing.

- By convention, controller classes:

  - Reside in the project's root-level Controllers folder.
  - Inherit from Microsoft.AspNetCore.Mvc.Controller.
  - A controller is an instantiable class in which at least one of the following conditions is true:

  - The class name is suffixed with Controller.
  - The class inherits from a class whose name is suffixed with Controller.
  - The [Controller] attribute is applied to the class.

ASP.NET Core DI - Views

- ASP.NET Core supports dependency injection into views. This can be useful for view-specific services, such as localization or data required only for populating view elements.
- You should try to maintain separation of concerns between your controllers and views. Most of the data your views display should be passed in from the controller.

Demonstração

- Demo - MVC

Exercise - MVC

- Create controller :
  - Model Movies
- Create a View

  - Implement CRUD actions (actions and views)

Exercise - Security

- Create a controller UserController
  - Implement Login Page
  - Implement a Register Page
  - Apply a restrition:
    - Only registered users can add movies

Unit test controller logic

- Unit Tests involve testing a part of an app in isolation from its infrastructure and dependencies.
- When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Demonstration

- Demo - Tests

Exercício

- Create Unit Tests:
  - Movies Controller