

.NET Platform

Especialização Full Stack Development

Prof. Dr. João Ricardo Favan

10 de Setembro de 2022

Agenda

- Cronogramas das Proximas Aulas
- Clean Architecture
- Padrões de Projetos
- Organização de Recursos
- Autenticação e Autorização
- Implantação

10/09/2022

Desenvolvimento de
aplicações com Clean
Architecture

Aplicações MVC
Aplicações WebAPI

17/09/2022

Persistência de Dados
Testes de unidade e
integração

Soluções em dotnet

24/09/2022

Machine Learning em
ambiente dotnet

Treinamento e
utilização de machine
learning

Objetivo

- Conhecer o modelo de arquitetura Clean Architecture par .NET
- Desenvolver aplicações no padrão MVC
- Desenvolver aplicações com API

MVC e Razor Pages

O termo MVC significa "Model-View-Controller", um padrão de interface do usuário que divide a responsabilidade de responder às solicitações do usuário em várias partes.

MVC e Razor Pages

As Razor Pages são integradas e usam os mesmos recursos para roteamento, model binding, filtros, autorização etc.

Não tem pastas e arquivos separados para controladores, modelos, exibições etc. e usam o roteamento baseado em atributo.

Por que utilizar Razon Pages?

Elas oferecem uma maneira mais simples de criar recursos de aplicativo baseados em página, como formulários que não são de SPA

Por que utilizar Razon Pages?

O modelo de página de uma Razor Page combina as responsabilidades de um controlador MVC e de um viewmodel. Em vez de manipular as solicitações com métodos de ação do controlador, são executados manipuladores de modelo de página, como "OnGet()", renderizando suas próprias páginas associadas por padrão.

Mapeando Solicitações de Respostas

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "default", pattern:
"{controller=Home}/{action=Index}/{id?}");
});
```

definidas no código, especificando as convenções de roteamento com uma sintaxe

Mapeando Solicitações de Respostas

```
[Route("Home")]  
public class HomeController : Controller  
{  
    [Route("")] // Combines to define the route template "Home"  
    [Route("Index")] // Combines to define route template "Home/Index"  
    [Route("/")] // Does not combine, defines the route template ""  
    public IActionResult Index() {}  
}
```

As rotas de atributo são aplicadas aos controladores e às ações diretamente, em vez de serem especificadas globalmente. Essa abordagem tem a vantagem de facilitar muito mais a descoberta delas quando você está analisando um método específico, mas significa que as informações de roteamento não são mantidas em um só lugar no aplicativo

Mapeando Solicitações de Respostas

```
[Route("[controller]")]  
public class ProductsController : Controller  
{  
    [Route("")] // Matches 'Products'  
    [Route("Index")] // Matches 'Products/Index'  
    public IActionResult Index() {}  
}
```

Mapeando Solicitações de Respostas

As Razor Pages não usam o roteamento de atributo. Podemos especificar informações de modelo de rota adicionais para uma Razor Page como parte de sua diretiva @page:

```
@page "{id:int}"
```

```
/Products/123
```

Manter os controladores sob controle

Cada página individual recebe os próprios arquivos e classes dedicadas apenas para seus manipuladores

Essas classes naturalmente aumentavam para conter várias responsabilidades e dependências, o que dificultava a manutenção

Cada ação usa a instância do mediador para enviar uma mensagem, que é processada por um manipulador específico daquela ação e só precisa das dependências exigidas por essa ação

Manter os controladores sob controle

```
public class OrderController : Controller
{
    private readonly IMediator _mediator;

    public OrderController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet]
    public async Task<IActionResult> MyOrders()
    {
        var viewModel = await _mediator.Send(new GetMyOrders(User.Identity
        return View(viewModel);
    }
    // other actions implemented similarly
}
```




```
public class GetMyOrdersHandler : IRequestHandler<GetMyOrders, IEnumerable<OrderViewModel>>
{
    private readonly IOrderRepository _orderRepository;
    public GetMyOrdersHandler(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public async Task<IEnumerable<OrderViewModel>> Handle(GetMyOrders request, CancellationToken cancellationToken)
    {
        var specification = new CustomerOrdersWithItemsSpecification(request.Specification);
        var orders = await _orderRepository.ListAsync(specification);
        return orders.Select(o => new OrderViewModel
        {
            OrderDate = o.OrderDate,
            OrderItems = o.OrderItems?.Select(oi => new OrderItemViewModel
            {
                PictureUrl = oi.ItemOrdered.PictureUri,
                ProductId = oi.ItemOrdered.CatalogItemId,
                ProductName = oi.ItemOrdered.ProductName,
                UnitPrice = oi.UnitPrice,
                Units = oi.Units
            }).ToList(),
            OrderNumber = o.Id,
            ShippingAddress = o.ShipToAddress,
            Total = o.Total()
        });
    }
}
```

Mapeando Solicitações de Respostas

O resultado final dessa abordagem é que os controladores são muito menores e se concentram principalmente no roteamento e no model binding, enquanto os manipuladores individuais são responsáveis pelas tarefas específicas necessárias para um determinado ponto de extremidade.

Declaração das Dependências

Para aplicações mais simples, estas podem ser feitas diretamente no arquivo *Program.cs*

```
var builder = WebApplication.CreateBuilder(args);  
  
// Add services to the container.  
builder.Services.AddRazorPages();  
  
var app = builder.Build();
```

Declaração das Dependências

O arquivo Startup.cs é configurado para dar suporte à Injeção de Dependência

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
    }
}
```

Organização dos Recursos

Os aplicativos ASP.NET Core organizam sua estrutura de pastas para incluir Controladores e Exibições e, frequentemente, ViewModels. Em geral, o código do lado do cliente para dar suporte a essas estruturas do lado do servidor é armazenado separadamente na pasta `wwwroot`

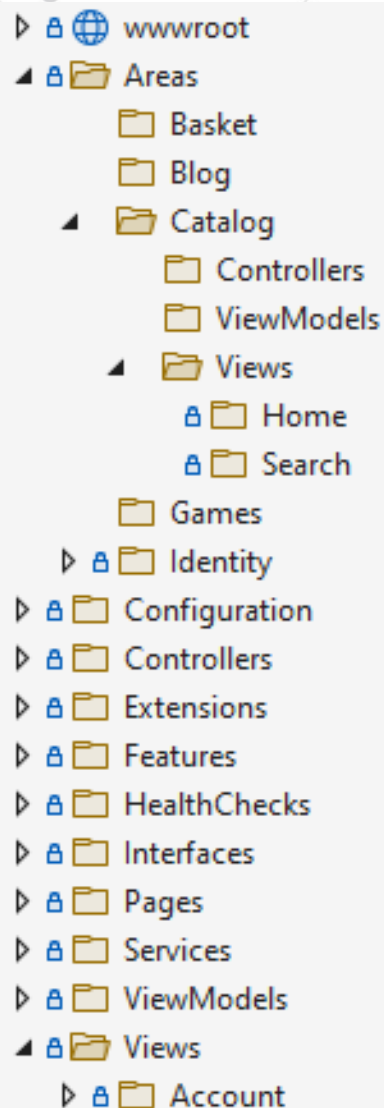
Organização dos Recursos










































Uma solução para esse problema é organizar o código do aplicativo por recurso, em vez de por tipo de arquivo. Esse estilo organizacional é normalmente conhecido como pastas de recurso ou fatias de recurso

Na plataforma .Net essas fatias de recursos são chamadas de *areas*

Organização dos Recursos

```
[Area("Catalog")]  
public class HomeController  
{  
}
```



- ▶   wwwroot
- ▶   Areas
 -  Basket
 -  Blog
 - ▶   Catalog
 -  Controllers
 -  ViewModels
 - ▶   Views
 -   Home
 -   Search
 -  Games
 - ▶   Identity
- ▶   Configuration
- ▶   Controllers
- ▶   Extensions
- ▶   Features
- ▶   HealthChecks
- ▶   Interfaces
- ▶   Pages
- ▶   Services
- ▶   ViewModels
- ▶   Views
- ▶   Account

APIs e aplicativos Blazor

Se o aplicativo incluir um conjunto de APIs Web, que precisa ser protegido, essas APIs deverão ser configuradas como um projeto separado do aplicativo de exibição ou de Razor Pages

Também é muito provável que eles adotem mecanismos diferentes de segurança, com aplicativos baseados em formulários padrão que aproveitam a autenticação baseada em cookie e APIs com maior probabilidade de usar a autenticação baseada em token

Interesses paralelos

Conforme os aplicativos crescem, fica cada vez mais importante excluir interesses paralelos para eliminar a duplicação e manter a consistência.

Alguns exemplos de interesses paralelos são autenticação, regras de validação de modelos, cache de saída e tratamento de erro, embora haja muitos outros

Interesses paralelos

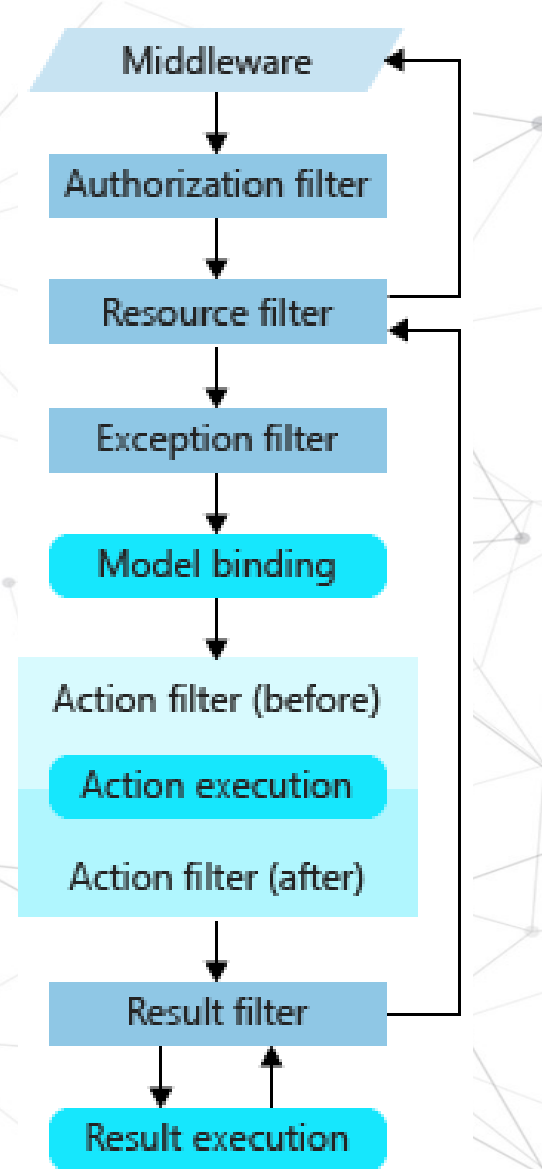
um filtro pode ser executado antes e após o model binding, antes e após uma ação ou antes e após o resultado de uma ação

Os filtros são implementados como atributos, de modo que possa aplicá-los aos controladores ou às ações

Os filtros especificados no nível da ação substituem ou se baseiam nos filtros especificados no nível do controlador e substituem os filtros globais

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous] // overrides the Authorize attribute
    public async Task<IActionResult> Login() {}
    public async Task<IActionResult> ForgotPassword() {}
}
```

```
[HttpPut("{id}")]
public async Task<IActionResult> Put(int id, [FromBody]Author author)
{
    if ((await _authorRepository.ListAsync()).All(a => a.Id != id))
    {
        return NotFound(id);
    }
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    author.Id = id;
    await _authorRepository.UpdateAsync(author);
    return Ok();
}
```



Segurança e Autenticação

O ASP.NET Core Identity é um sistema de associação que pode ser usado para dar suporte à funcionalidade de login para o aplicativo. Ele tem suporte para contas de usuário local, bem como suporte para provedores de login externo de provedores como a conta da Microsoft, Twitter, Facebook, Google e muito mais.

Configurar o Identity em *Program.cs*

Configura serviços da instância *WebHostBuilder* e, depois que o aplicativo é criado, configura-se o *middleware*.

Os pontos principais a serem observados são a chamada a *AddDefaultIdentity* para os serviços necessários e as chamadas *UseAuthentication* e *UseAuthorization* que adicionam o *middleware* necessário.

Autenticação

Na autenticação baseada na Web, normalmente até cinco ações podem ser executadas no decorrer da autenticação do cliente de um sistema. Eles são:

Autenticar. Usar as informações fornecidas pelo cliente para criar uma identidade a ser usada no aplicativo.

Desafio. Essa ação é usada para exigir que o cliente se identifique.

Autenticação

Na autenticação baseada na Web, normalmente até cinco ações podem ser executadas no decorrer da autenticação do cliente de um sistema. Eles são:

Proibir. Informar ao cliente que ele está proibido de executar uma ação.

Entrar. Persistir o cliente existente de alguma forma.

Sair. Remover o cliente da persistência

Autenticação de APIs

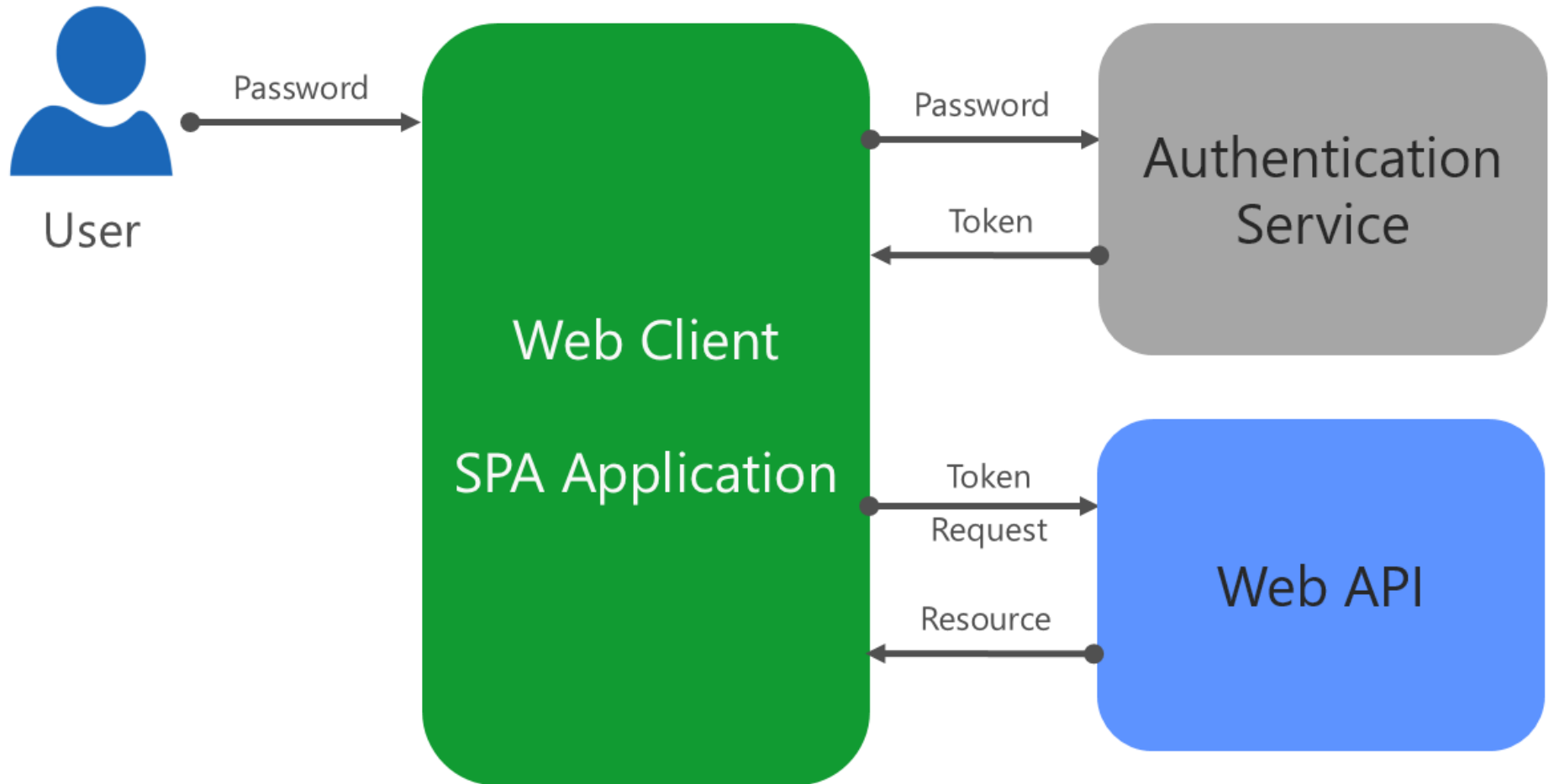
```
builder.Services
    .AddAuthentication(config =>
    {
        config.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(config =>
    {
        config.RequireHttpsMetadata = false;
        config.SaveToken = true;
        config.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(key),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
```

Autenticação de APIs

A forma mais simples de autorização envolve a restrição do acesso a usuários anônimos. Essa funcionalidade pode ser obtida aplicando o atributo [Authorize] a determinados controladores ou ações

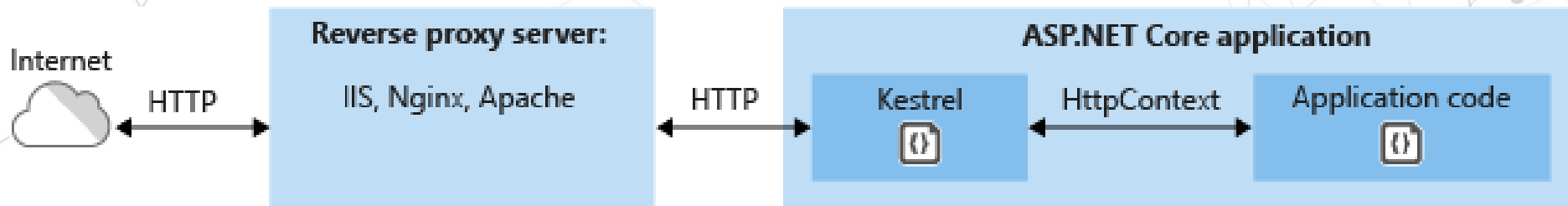
```
[Authorize(Roles = "HRManager,Finance")]  
public class SalaryController : Controller  
{  
  
}
```

Token-Based Authentication



Implantação

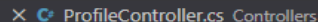

Há algumas etapas envolvidas no processo de implantação do aplicativo ASP.NET Core, independentemente do local em que ele será hospedado



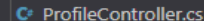




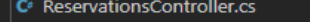
EXPLORER

X  ProfileController.cs Controllers

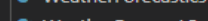
1. *Controlled* – The controller is a person or organization that has the authority to direct and manage the operations of the system.



C# ProfileController.cs



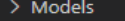
C# ReservationsController.cs



- Weather Forecasts
- Weather Forecasting



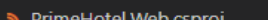
> Data



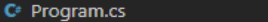
> Models



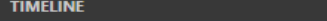
7 Properties



PrimeHotel Web csproj



C# Program.cs



TIMELINE

C# ProfileController.cs X

Controllers > ProfileController.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Data;
4  using System.Diagnostics;
5  using System.Threading.Tasks;
6  using Bogus;
7  using Microsoft.AspNetCore.Mvc;
8  using Microsoft.Data.SqlClient;
9  using Microsoft.EntityFrameworkCore;
10 using Microsoft.Extensions.Configuration;
11 using PrimeHotel.Web.Models;
12
13 namespace PrimeHotel.Web.Controllers
14 {
15     [ApiController]
16     [Route("[controller]")]
17     public class ProfileController : ControllerBase
18     {
19         private readonly PrimeDbContext primeDbContext;
20         private readonly string connectionString;
21
22         public ProfileController(PrimeDbContext _primeDbContext, IConfiguration _configuration)
23         {
24             connectionString = _configuration.GetConnectionString("HotelDB");
25             primeDbContext = _primeDbContext;
26         }
27
28         [HttpGet]

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

C#

```

Downloading package '.NET Core Debugger (Windows / x64)' (42010 KB)..... Done!
Validating download...
Integrity Check succeeded.
Installing package '.NET Core Debugger (Windows / x64)'

```

```
Downloading package 'Razor Language Server (Windows / x64)' (51084 KB)..... Done!
Installing package 'Razor Language Server (Windows / x64)'
```

Finished