AspNetCore
REST API

Prof. Me. Hélio Lourenço
Esperidião Ferreira

AspNetCore.Mvc

- AspNetCore.Mvc é o framework de desenvolvimento web do ASP.NET Core, responsável por implementar o padrão MVC (Model-View-Controller).
  - Fornece todos os **recursos essenciais** para construir **APIs REST** e **aplicações web** em C#, incluindo:
  - **Controladores (Controllers)** — classes que recebem e tratam requisições HTTP;
  - **Modelos (Models)** — classes que representam os dados e regras de negócio;
  - **Views (Views)** — páginas HTML dinâmicas (usadas em aplicações MVC, não em APIs);
  - **Roteamento (Routing)** — define como as URLs são mapeadas para métodos do controlador;
  - **Validação e Model Binding** — converte e valida automaticamente dados vindos de requisições;
  - **Filtros (Filters)** — executam lógica antes ou depois das ações do controlador.

Kestrel

- O Kestrel é o servidor web interno usado pelo ASP.NET Core para lidar com requisições HTTP. Ele é um servidor web multiplataforma e de alto desempenho, sendo o servidor web padrão para aplicações ASP.NET Core. O Kestrel é utilizado principalmente para hospedar a aplicação web e gerenciar a comunicação entre a aplicação e os clientes, como navegadores ou APIs externas.
- Características do Kestrel:
  - **Alta performance**: O Kestrel é projetado para ser rápido e eficiente, sendo capaz de lidar com milhares de conexões simultâneas.
  - **Multiplataforma**: Ele funciona em várias plataformas, incluindo Windows, Linux e macOS.
  - **Escuta requisições HTTP/HTTPS**: O Kestrel gerencia as requisições HTTP e HTTPS que chegam à aplicação, repassando essas requisições para o pipeline de middleware do ASP.NET Core.
  - **Suporte a SSL/TLS**: Ele pode ser configurado para servir conteúdos via HTTPS usando certificados SSL/TLS, oferecendo segurança nas comunicações.
  - **Configuração de portas e IPs**: O Kestrel permite configurar em quais portas e endereços IP a aplicação deve escutar

Ambiente de Desenvolvimento Robusto

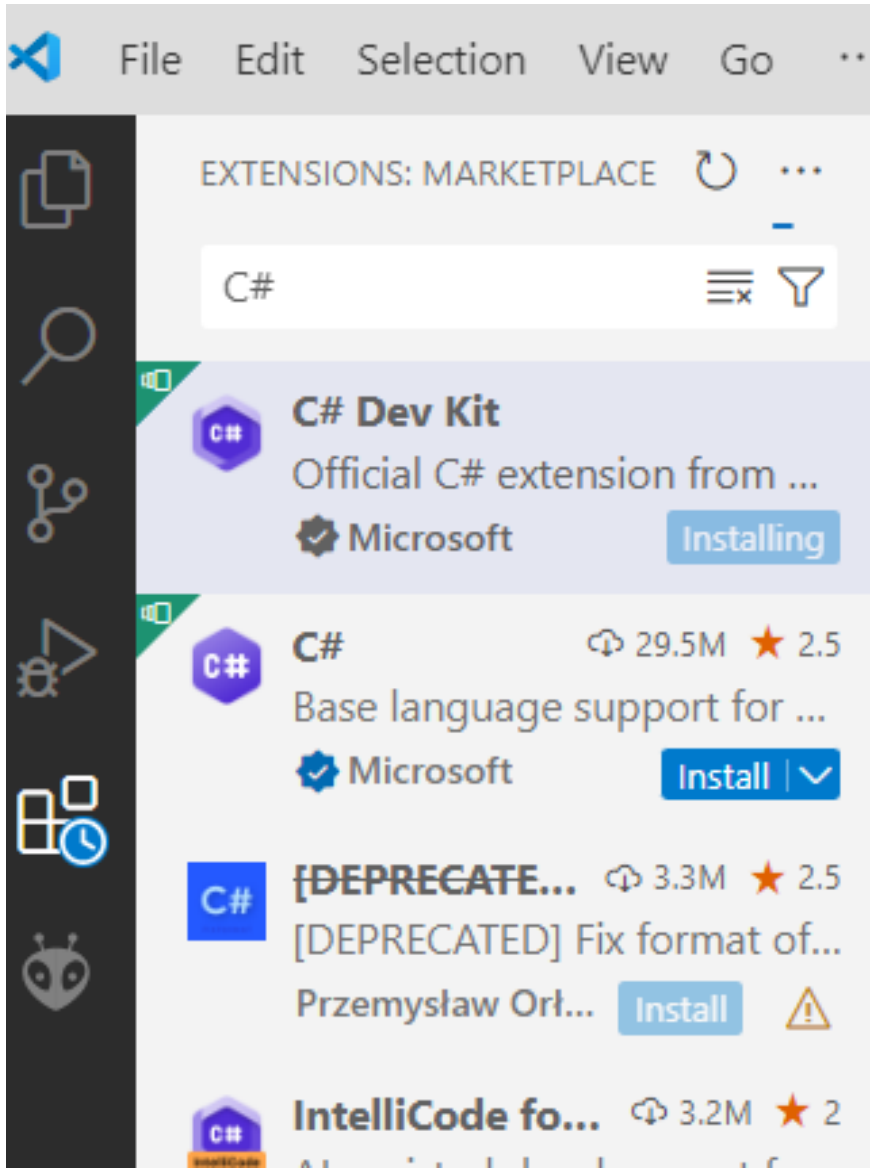
- **Visual Studio**: O C# possui um dos IDEs mais avançados, o Visual Studio, que oferece ferramentas poderosas para depuração, testes e desenvolvimento.
- **IntelliSense**: O suporte à autocompletação de código facilita a escrita e a manutenção do código.
- **Bibliotecas e Pacotes**: O C# e o ASP.NET Core têm um ecossistema rico com bibliotecas e pacotes disponíveis através do NuGet, facilitando a integração de funcionalidades adicionais.
- **Suporte a Microserviços**: O ASP.NET Core oferece suporte robusto para a criação de microserviços, permitindo que desenvolvedores construam sistemas distribuídos.
- **Recursos de Segurança Integrados**: O ASP.NET Core possui recursos integrados para autenticação e autorização, como suporte para JWT (JSON Web Tokens) e OAuth.
- **Proteção Contra Vulnerabilidades**: O framework possui proteções contra as principais vulnerabilidades de segurança, como injeções de SQL e XSS.
- **Compatibilidade com Azure**: C# se integra facilmente com serviços da nuvem da Microsoft, como o Azure, facilitando a implantação e escalabilidade de aplicações.
- **Interoperabilidade**: O C# pode interagir facilmente com outras tecnologias da Microsoft, como SQL Server e serviços de mensageria.
- **Programação Assíncrona**: O C# oferece suporte nativo para programação assíncrona, permitindo que aplicações lidem com operações de entrada/saída sem bloquear threads, o que é fundamental para aplicações de rede.
- **Suporte a Padrões de Projeto**: C# e ASP.NET Core incentivam o uso de padrões de projeto como MVC (Model-View-Controller) e Repository, que ajudam a manter o código organizado e fácil de manter

Curva de Aprendizado

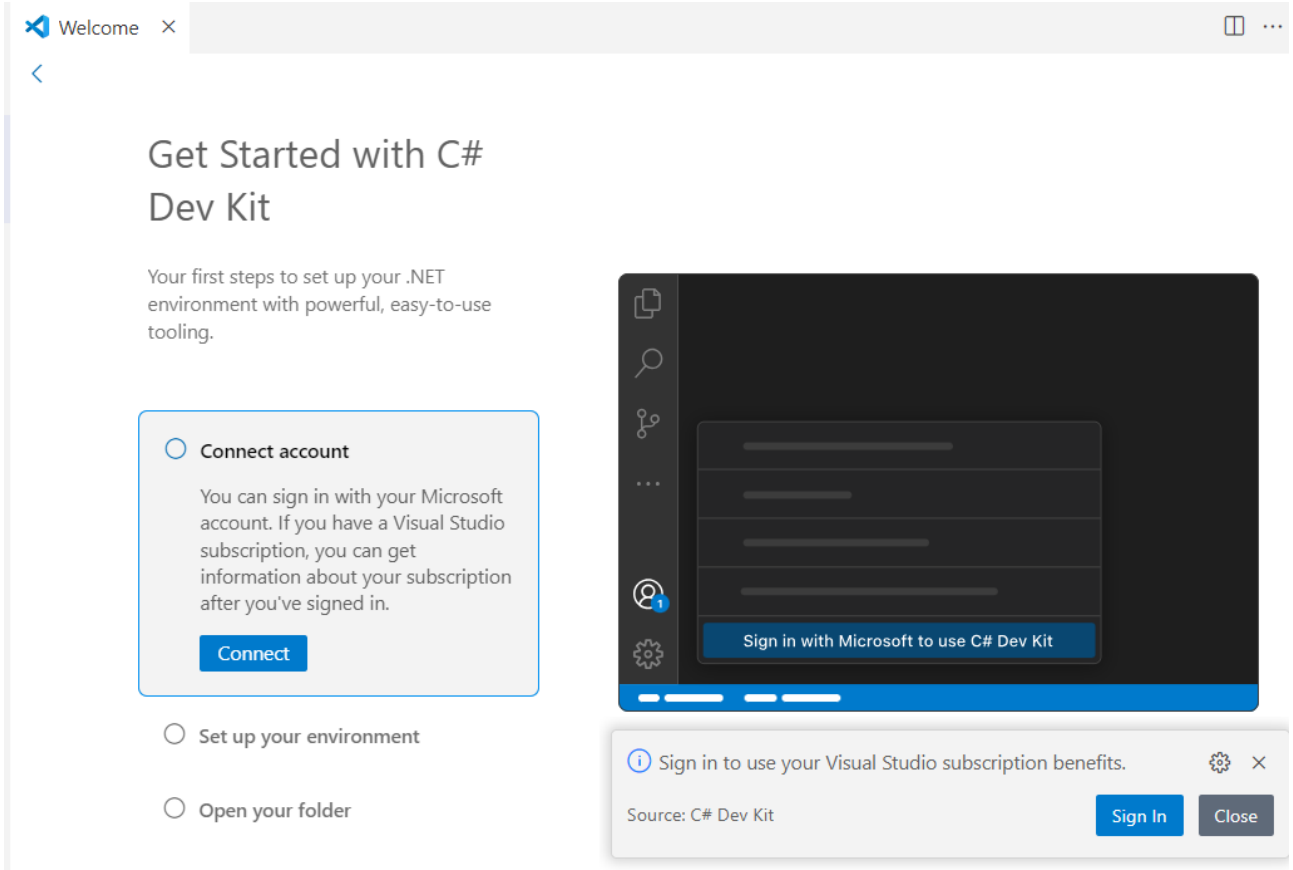
- **Complexidade**: C# pode ser considerado mais complexo para iniciantes em comparação com linguagens como Python ou JavaScript, especialmente devido à sua vasta gama de funcionalidades e conceitos avançados (como programação assíncrona, LINQ, etc.).
- ***Frameworks***: A variedade de frameworks e bibliotecas disponíveis pode ser confusa para novos desenvolvedores que estão começando.
- **Ambiente de Desenvolvimento**: Embora o .NET Core seja gratuito e de código aberto, algumas ferramentas de desenvolvimento e ambientes, como o Visual Studio (especialmente as versões mais completas), podem ter custos associados. Isso pode ser uma preocupação para pequenas empresas ou projetos de baixo orçamento.
- **Windows**: C# e .NET tradicionalmente têm sido mais associados ao ecossistema Windows, o que pode ser uma limitação em ambientes onde outras plataformas (como Linux ou macOS) são preferidas. No entanto, o .NET Core e o .NET 5+ têm abordado essa questão, permitindo que aplicativos sejam executados em várias plataformas.
- **Adoção em Web e Mobile**: Embora C# e ASP.NET sejam populares para desenvolvimento de back-end, outras linguagens como php e JavaScript (com Node.js) têm dominado o desenvolvimento web, enquanto linguagens como Java e Swift são mais comuns no desenvolvimento mobile. Isso pode resultar em uma comunidade menor e menos recursos disponíveis em algumas áreas.

C# Dev Kit extension

- Instale a extensão:
  - C# dev Kit

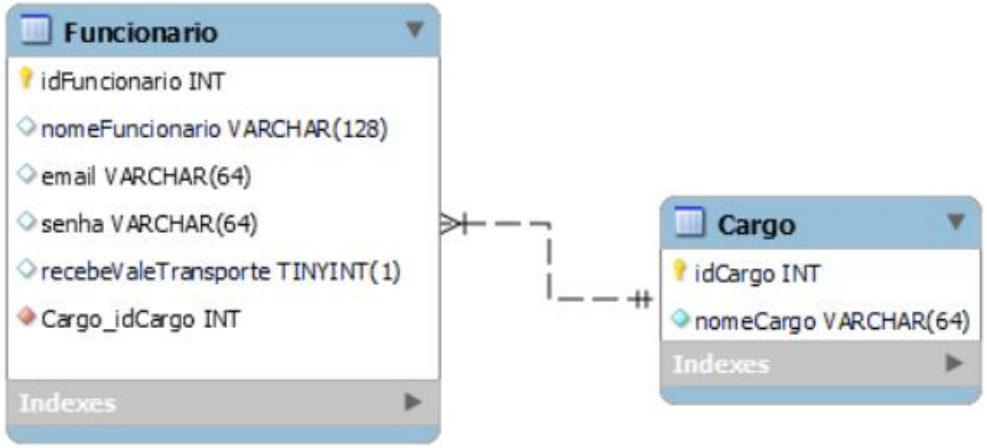


Fazer login após a instalação é opcional.



Exemplo CRUD

- Utilize como base o projeto: https://github.com/helioesperidiao/api_csharp_funcionario_cargo



Para exportar(Database >> Foward Engineer) o modelo para os códigos sql de criação é nessário que seja mudada a versão do mysql para 5.5.6.
Vá em : Edit>> preferences >> mysql >> default target Mysql version: 5.5.6

Novo projeto

- Crie um novo projeto
- Crie uma pasta no Windows e abra no visual studio code
- Abra o terminal dentro do visual stuido code e digite:
  - dotnet new webapi -n Api
- cd Api
- Md Modelo
- Md Controle

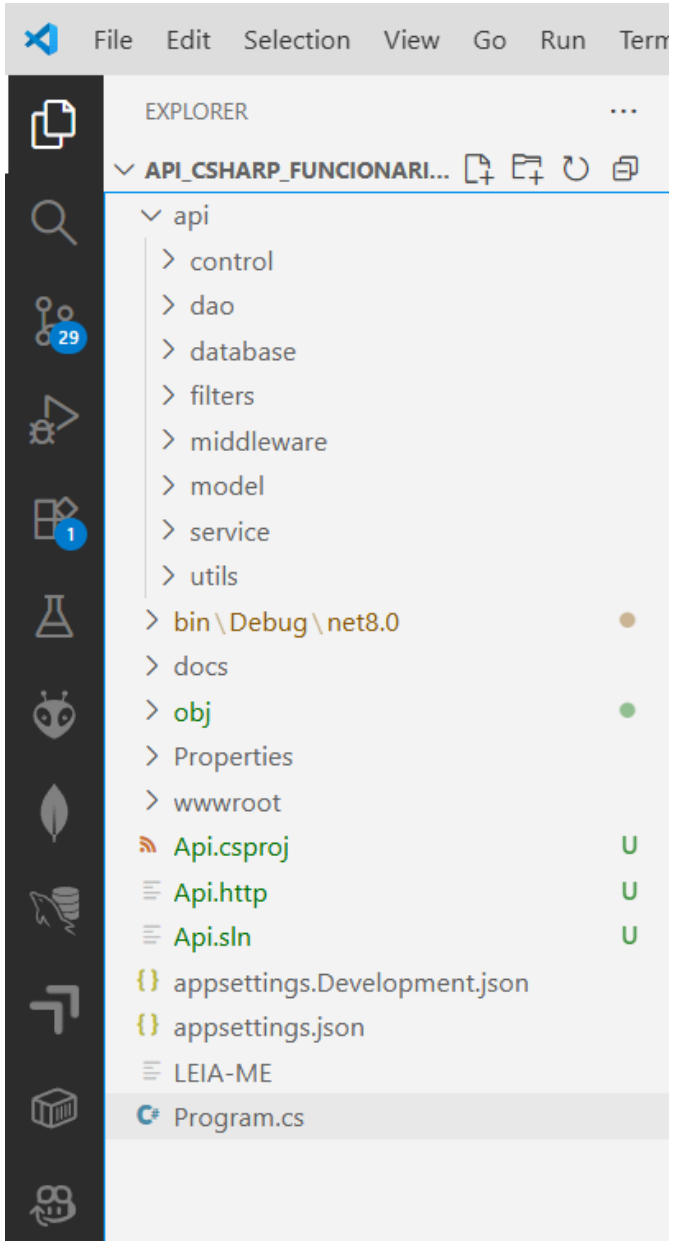
Estrutura de diretórios:

- Api
  - Control
  - dao
  - database
  - filter(aplicados a cada rota)
  - middleware(aplicados a todas as rotas)
  - model
  - service
  - utils

Arquivos estáticos:

wwwroot

Esta pasta é a pasta dos arquivos estáticos, ou seja html, css, js, etc



Pacotes

- Instalar o pacote de acesso ao mysql
- **Adicionar o Pacote MySQL:**
  - dotnet add package MySql.Data
- **Pacote de criptografia** (salvar senha)
  - dotnet add package BCrypt.Net-Next
- **Pacote de autenticação jwt** (validar/gerar tokens)
  - dotnet add package System.IdentityModel.Tokens.Jwt

Modelo

O Model em MVC representa os dados da aplicação e também pode conter regras de domínio.

Regras de domínio são as regras que definem como os dados podem existir ou interagir, independente de banco de dados ou interface.

O Model não deve se preocupar com persistência nem com apresentação.

```
using System;
namespace Api.Model{
        public class Cargo{
        // Atributos privados
        private int _idCargo;
        private string _nomeCargo = string.Empty;
        public Cargo(){//Console.WriteLine("⬆️  Cargo.constructor()");
        }
        public int IdCargo{
            get => this._idCargo;
            set{
                if (value <= 0){
                    throw new ArgumentException("IdCargo deve ser maior que zero.");
                }
                this._idCargo = value;
            }
        }
        public string NomeCargo{
            get => this._nomeCargo;
            set {
                if (string.IsNullOrWhiteSpace(value)){
                    throw new ArgumentException("NomeCargo não pode ser nulo ou vazio.");
                }
                string nome = value.Trim();
                if (nome.Length < 3){
                    throw new ArgumentException("NomeCargo deve ter pelo menos 3 caracteres.");
                }
                if (nome.Length > 64){
                    throw new ArgumentException("NomeCargo deve ter no máximo 64 caracteres.");
                }
                this._nomeCargo = nome;
            }
        }
    }
}
```

DAO

DAO é um padrão de projeto cuja função principal é isolar o acesso ao banco de dados da lógica de negócio da aplicação.

Ele atua como uma camada de abstração entre seu Model e o banco de dados.

Isso significa que o restante da aplicação não precisa saber como os dados são armazenados (SQL, NoSQL, arquivos etc.).

Responsabilidades do DAO

- Um DAO típico faz:
  - CRUD (Create, Read, Update, Delete) de um Model:
  - create(obj) → insere o objeto no banco.
  - read(id) → retorna um objeto a partir do banco.
  - update(obj) → atualiza dados existentes.
  - delete(id) → remove do banco.

```
public class CargoDAO
   {
       private readonly MySqlDatabase _database;


       /// <summary>
       /// Construtor do DAO, recebe a instância de MySqlDatabase.
       /// </summary>
       /// <param name="databaseInstance">Instância de MySqlDatabase injetada.</param>
       public CargoDAO(MySqlDatabase databaseInstance)
       {
           Console.WriteLine("⬆️  CargoDAO.constructor()");
           _database = databaseInstance ?? throw new ArgumentNullException(nameof(databaseInstance));
       }
```

```
public async Task<int> Create(Cargo objCargoModel)
        {
            Console.WriteLine("🟢 CargoDAO.Create()");
            string SQL = "INSERT INTO cargo (nomeCargo) VALUES (@nomeCargo);";

            await using var conn = await _database.GetConnection();
            await using var cmd = new MySqlCommand(SQL, conn);
            cmd.Parameters.AddWithValue("@nomeCargo", objCargoModel.NomeCargo);

            int insertedId = 0;

            await cmd.ExecuteNonQueryAsync();
            insertedId = (int)cmd.LastInsertedId;

            if (insertedId <= 0)
            {
                throw new Exception("Falha ao inserir cargo");
            }

            return insertedId;
        }
```

```
    public async Task<bool> Delete(Cargo objCargoModel)
        {
            Console.WriteLine("🟢 CargoDAO.Delete()");


            string SQL = "DELETE FROM cargo WHERE idCargo = @idCargo;";


            await using var conn = await _database.GetConnection();
            await using var cmd = new MySqlCommand(SQL, conn);
            cmd.Parameters.AddWithValue("@idCargo", objCargoModel.IdCargo);


            int affectedRows = await cmd.ExecuteNonQueryAsync();
            return affectedRows > 0;
        }
```

```
public async Task<bool> Update(Cargo objCargoModel){
            Console.WriteLine(" CargoDAO.Update()");

            string SQL = "UPDATE cargo SET nomeCargo = @nomeCargo WHERE idCargo = @idCargo;";

            await using var conn = await _database.GetConnection();
            await using var cmd = new MySqlCommand(SQL, conn);
            cmd.Parameters.AddWithValue("@nomeCargo", objCargoModel.NomeCargo);
            cmd.Parameters.AddWithValue("@idCargo", objCargoModel.IdCargo);

            int affectedRows = await cmd.ExecuteNonQueryAsync();
            return affectedRows > 0;
        }
```

```
public async Task<List<Cargo>> FindAll(){
            Console.WriteLine(" CargoDAO.FindAll()");
            string SQL = "SELECT * FROM cargo;";

            List<Cargo> result = new List<Cargo>();

            await using var conn = await _database.GetConnection();
            await using var cmd = new MySqlCommand(SQL, conn);
            await using var reader = await cmd.ExecuteReaderAsync();

            while (await reader.ReadAsync()){
                Cargo linha = new Cargo();
                linha.IdCargo = reader.GetInt32("idCargo");
                linha.NomeCargo = reader.GetString("nomeCargo");
                result.Add(linha);
            }
            return result;
        }
```

```
public async Task<Cargo?> FindById(int idCargo){
            Console.WriteLine("  CargoDAO.FindById()");
            var results = await FindByField("idCargo", idCargo);
            return results.Count > 0 ? results[0] : null;
        }
```

```
public async Task<List<Cargo>> FindByField(string field, object value){
            Console.WriteLine($"🟢 CargoDAO.FindByField() - Campo: {field}, Valor: {value}");

            var allowedFields = new HashSet<string> { "idCargo", "nomeCargo" };
            if (!allowedFields.Contains(field))
                throw new ArgumentException($"Campo inválido para busca: {field}");

            string SQL = $"SELECT * FROM cargo WHERE {field} = @value;";

            List<Cargo> result = new List<Cargo>();

            await using var conn = await _database.GetConnection();
            await using var cmd = new MySqlCommand(SQL, conn);
            cmd.Parameters.AddWithValue("@value", value);

            await using var reader = await cmd.ExecuteReaderAsync();
            while (await reader.ReadAsync())
            {
                Cargo linha = new Cargo();
                linha.IdCargo = reader.GetInt32("idCargo");
                linha.NomeCargo = reader.GetString("nomeCargo");
                result.Add(linha);
            }

            return result;
        }
```

service

- Um Service é uma camada intermediária entre o Controller e o DAO/Model.
  - Sua função principal é: Encapsular regras de negócio complexas que não cabem apenas no Model.
  - Orquestrar operações envolvendo múltiplos DAOs ou Models.
  - Garantir consistência antes de persistir ou retornar dados.
- **Por que usar Service?**
  - Evita que o **Controller fique cheio de lógica**.
  - Evita que o **DAO fique com regras de negócio**.
  - Deixa o código mais modular e fácil de testar.
- O Service não conversa diretamente com a camada de apresentação (View).
- Ele responde apenas ao Controller, que é o responsável por receber o input do usuário e decidir o que mostrar na interface.
- O Service aplica regras de negócio e orquestra DAOs.

```
public class CargoService
    {
        private readonly CargoDAO _cargoDAO;


        /// <summary>
        /// Construtor com injeção de dependência do DAO.
        /// </summary>
        /// <param name="cargoDAODependency">Instância de CargoDAO</param>
        public CargoService(CargoDAO cargoDAODependency)
        {
            Console.WriteLine("⬆️  CargoService.constructor()");
            _cargoDAO = cargoDAODependency ?? throw new ArgumentNullException(nameof(cargoDAODependency));
        }
```

```
public async Task<int> CreateCargo(string nomeCargo){
            Console.WriteLine("🟣 CargoService.CreateCargoAsync()");

            Cargo cargo = new Cargo();
            cargo.NomeCargo = nomeCargo; // valida regra de domínio no set

            // Valida regra de negócio: não permitir nomes duplicados
            List<Cargo> resultado = await _cargoDAO.FindByField("nomeCargo", cargo.NomeCargo);

            if (resultado.Count > 0)
            {
                throw new ErrorResponse(
                    400,
                    "Cargo já existe",
                    $"O cargo {cargo.NomeCargo} já existe"
                );
            }

            return await _cargoDAO.Create(cargo);
        }
```

```
public async Task<List<Cargo>> FindAll()
        {
            Console.WriteLine("🟣 CargoService.FindAll()");
            return await _cargoDAO.FindAll();
        }
```

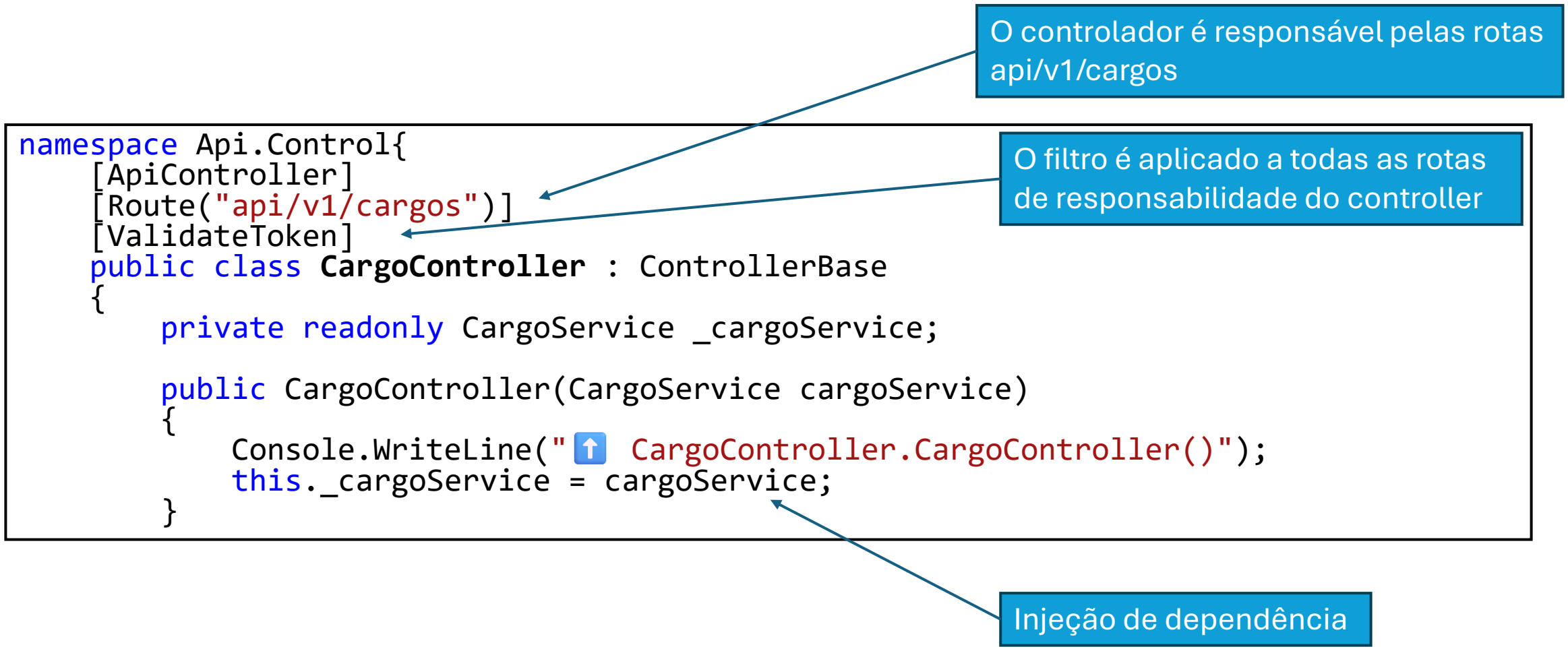
```
        /// <summary>
        /// Retorna um cargo por ID
        /// </summary>
        public async Task<Cargo?> FindById(int idCargo){
            Console.WriteLine("  CargoService.FindById()");

            Cargo cargo = new Cargo();
            cargo.IdCargo= idCargo; // valida regra de domínio no set

            return await _cargoDAO.FindById(cargo.IdCargo);
        }
```

```
        /// <summary>
        /// Atualiza um cargo existente
        /// </summary>
        public async Task<bool> UpdateCargo(int idCargo, string nomeCargo){
            Console.WriteLine("  CargoService.UpdateCargo()");

            Cargo cargo = new Cargo();
            cargo.IdCargo = idCargo;     // valida regra de domínio no set
            cargo.NomeCargo = nomeCargo; // valida regra de domínio no set

            return await _cargoDAO.Update(cargo);
        }
```

```
        public async Task<bool> DeleteCargo(int idCargo){
            Console.WriteLine("  CargoService.DeleteCargo()");

            Cargo cargo = new Cargo();
            cargo.IdCargo = idCargo;    // valida regra de domínio no set
            return await _cargoDAO.Delete(cargo);
        }
```

Controle

- O **Controller** é a **camada intermediária entre a View e o Service**.
  Ele é responsável por:
- **Receber input do usuário**
  - requisição HTTP (GET, POST, PUT, DELETE)
- **Interpretar a ação do usuário**
  - Saber qual operação deve ser feita, como criar, atualizar, ou buscar dados.
- **Chamar o Service**
  - Envia os dados recebidos para as regras de negócio (Service)
- **Receber o resultado**
  - O Controller pega o retorno do Service prepara a resposta.
- **Enviar resposta para a View**
  - Traduz o resultado da operação para a interface do usuário: mensagem (JSON).


O controlador é responsável pelas rotas api/v1/cargos
O filtro é aplicado a todas as rotas de responsabilidade do controller
namespace Api.Control{
[ApiController]
[Route("api/v1/cargos")]
[ValidateToken]
public class CargoController : ControllerBase
{
private readonly CargoService _cargoService;
public CargoController(CargoService cargoService)
{
Console.WriteLine("⬆️ CargoController.CargoController()");
this._cargoService = cargoService;
}
Injeção de dependência

Método responsável por reposponder requisições get na rota: api/v1/cargos

```
[HttpGet]
 public async Task<IActionResult> Index(){
     Console.WriteLine("🔵 CargoController.Index()");
     List<Cargo> listaCargos = await _cargoService.FindAll();

     object resposta = new
     {
         success = true,
         message = "Busca realizada com sucesso",
         data = new { cargos = listaCargos }
     };

     return Ok(resposta);


 }
```

Antes de executar o método show()
o filtro ValidateCargoId é chamado.

```
// GET: api/v1/cargos/{id}
[HttpGet("{idCargo}")]
[ValidateCargoId]
public async Task<IActionResult> Show(int idCargo)
{
    Console.WriteLine("🔵 CargoController.Show()");
    Cargo? cargoEncontrado = await _cargoService.FindById(idCargo);

    List<Cargo> cargosArray = new List<Cargo>();


    if (cargoEncontrado != null){
        cargosArray = new List<Cargo> { cargoEncontrado };
    }

    object resposta = new{
        success = true,
        message = "Executado com sucesso",
        data = new { cargos = cargosArray }
    };
    return Ok(resposta);
}
```

```
// POST: api/v1/cargos
        [HttpPost]
        [ValidateCargoBody]
        public async Task<IActionResult> Store([FromBody] JsonElement requestBody){
            Console.WriteLine("🔵 CargoController.Store()");
            // Extrai o objeto "cargo" do corpo da requisição
            requestBody.TryGetProperty("cargo", out JsonElement cargoElem);
            // Extrai o nome do cargo
            string nomeCargo = cargoElem.GetProperty("nomeCargo").GetString() ?? "";
            // Cria o cargo e obtém o ID gerado
            int novoId = await _cargoService.CreateCargo(nomeCargo);
            // Monta o objeto de retorno
            Cargo novoCargo = new Cargo();
            novoCargo.IdCargo = novoId;
            novoCargo.NomeCargo = nomeCargo;
            object[] cargosArray = new object[] { novoCargo };

            object resposta = new  {
                success = true,
                message = "Cadastro realizado com sucesso",
                data = new { cargos = cargosArray }
            };
            // Retorna 201 Created com os dados do novo cargo
            return StatusCode(201, resposta);
        }
```

```
// PUT: api/v1/cargos/{id}
        [HttpPut("{idCargo}")]
        [ValidateCargoBody]
        public async Task<IActionResult> Update(int idCargo, [FromBody] JsonElement requestBody){
            Console.WriteLine("🔵 CargoController.Update()");
            requestBody.TryGetProperty("cargo", out JsonElement cargoElem);
            string nomeCargo = cargoElem.GetProperty("nomeCargo").GetString() ?? "";
            bool atualizou = await _cargoService.UpdateCargo(idCargo, nomeCargo);
            Cargo cargoAtualizado = new Cargo           {
                IdCargo = idCargo,
                NomeCargo = nomeCargo
            };
            object[] cargosArray = new object[] { cargoAtualizado };
            if (atualizou){
                object resposta = new
                    success = true,
                    message = "Atualizado com sucesso",
                    data = new { cargos = cargosArray }
                };
                return Ok(resposta);
            }else{
                object resposta = new{
                    success = false,
                    message = "Cargo não encontrado para atualização",
                    data = new { cargos = cargosArray }
                };
                return NotFound(resposta);
            }
        }
```

```
        // DELETE: api/v1/cargos/{id}
        [HttpDelete("{idCargo}")]
        public async Task<IActionResult> Destroy(int idCargo){
            Console.WriteLine("🔵 CargoController.Destroy()");
            bool excluiu = await _cargoService.DeleteCargo(idCargo);

            Cargo cargoExcluido = new Cargo{
                IdCargo = idCargo
            };

            object[] cargosArray = new object[] { cargoExcluido };

            if (excluiu) {
                return NoContent();
            }
            else{
                object resposta = new{
                    success = false,
                    message = "Cargo não encontrado para exclusão",
                    data = new { cargos = cargosArray }
                };

                return NotFound(resposta);
            }

        }
```

Program.cs

- É o arquivo de entrada que injeta as dependências, configura os controles e inicia a aplicação

```
// Importa os namespaces (bibliotecas) necessários
using Microsoft.AspNetCore.Mvc;     // Fornece recursos para criar controladores e endpoints de API
using Api.DAO;                      // Contém classes de acesso a dados (Data Access Object)
using Api.Service;                  // Contém regras de negócio e lógica de serviço
using Api.Database;                 // Contém a classe responsável pela conexão com o banco de dados
MySQL
using Api.Middleware;               // Contém middlewares personalizados, como tratamento de erros

// O "builder" é um objeto que prepara (constrói) a configuração da aplicação web.
// Ele define serviços, middlewares e comportamentos do servidor HTTP.
WebApplicationBuilder builder = WebApplication.CreateBuilder();

// ------------------------------------------------------------
// Configuração do Servidor Kestrel
// ------------------------------------------------------------
// O Kestrel é o servidor web interno do ASP.NET Core.
// Aqui estamos configurando para que ele escute **somente** na porta 8080.
// Isso é útil para garantir que a API esteja acessível em http://localhost:8080
builder.WebHost.ConfigureKestrel(options =>
{
    options.ListenAnyIP(8080); // O servidor escutará conexões em qualquer endereço IP da máquina
});
```

MIDDLEWARES em AspNetCore.Mvc

- Middlewares são **componentes de pipeline** que processam **todas as requisições** que passam pelo servidor, **antes e/ou depois** que chegam ao MVC.
- Eles são mais **“baixos”** no pipeline — próximos do servidor (Kestrel).

Filters em AspNetCore.Mvc

- Filtros são executados **dentro do MVC**, já **depois** que a requisição passou pelos *middlewares* e chegou ao *controller*.
- Eles servem para **interceptar a execução de actions** (métodos de controller).

```
using Microsoft.AspNetCore.Mvc.Filters;
using System.Text.Json;
using Api.Utils;
namespace Api.Filters{
    public class ValidateCargoBody : ActionFilterAttribute{
        public override void OnActionExecuting(ActionExecutingContext context){
            Console.WriteLine("🔷 ValidateCargoBody.OnActionExecuting()");
            if (!context.ActionArguments.TryGetValue("requestBody", out var bodyObj) || bodyObj == null){
                throw new ErrorResponse(400,
                    "O objeto 'cargo' é obrigatório!",new { detalhe = "Corpo da requisição ausente ou incorreto" });
            }
            // Converte para JsonElement
            if (bodyObj is not JsonElement json){
                throw new ErrorResponse(400,
                    "Formato inválido", new { detalhe = "O corpo da requisição não é um JSON válido" });
            }
            // Verifica propriedade "cargo"
            if (!json.TryGetProperty("cargo", out JsonElement cargoElem)){
                throw new ErrorResponse(400,
                    "O objeto 'cargo' é obrigatório!", new { detalhe = "Propriedade 'cargo' ausente" });
            }           // Verifica "nomeCargo"
            if (!cargoElem.TryGetProperty("nomeCargo", out JsonElement nomeElem) ||
                string.IsNullOrWhiteSpace(nomeElem.GetString())){
                throw new ErrorResponse(400,
                    "O campo 'nomeCargo' é obrigatório!",new { detalhe = "Campo vazio ou não informado" });
            }
        }
    }
}
```

Program.cs

```
// ------------------------------------------------------------
// Banco de Dados - MySQL
// ------------------------------------------------------------
// Aqui registramos a classe MySqlDatabase como um serviço Singleton.
// Isso significa que uma única instância dela será usada em toda a aplicação.
// Dentro dela, são passadas as informações de conexão com o banco MySQL.
builder.Services.AddSingleton<MySqlDatabase>(serviceProvider =>
{
    return new MySqlDatabase(
        host: "localhost",    // Endereço do servidor MySQL
        user: "root",         // Usuário do banco
        password: "",         // Senha do banco
        database: "gestao_rh",// Nome do banco de dados
        port: 3306,           // Porta padrão do MySQL
        connectionLimit: 10   // Limite de conexões simultâneas
    );
});
```

```
// ------------------------------------------------------------
// Configuração de CORS (Cross-Origin Resource Sharing)
// ------------------------------------------------------------
// O CORS é necessário para permitir que outros domínios (como uma aplicação web frontend)
// consigam acessar esta API. Sem isso, o navegador bloqueia as requisições.
// Aqui criamos uma política que permite **qualquer origem**, **qualquer método** e **qualquer
cabeçalho**.
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAll", policy =>
        policy.AllowAnyOrigin()
              .AllowAnyMethod()
              .AllowAnyHeader());
});
```

Singleton

- No contexto do ASP.NET Core (e da injeção de dependência em geral), singleton é um padrão de criação de objetos que garante que apenas uma instância de uma determinada classe será criada e compartilhada durante toda a execução da aplicação.

- builder.Services.**AddSingleton**<FuncionarioService>();

- Então todos os controladores que receberem FuncionarioService no construtor estarão usando a mesma instância, o que permite, por exemplo, compartilhar caches ou configurações internas sem reinicializar objetos toda hora.

```
// Registra dependencias no container DI
//registra dependencia de banco de dados
builder.Services.AddSingleton<MySqlDatabase>(serviceProvider =>{
    return new MySqlDatabase(
        host: "localhost",
        user: "root",
        password: "",
        database: "gestao_rh",
        port: 3306,
        connectionLimit: 10
    );
});

//registra dependencia de DAOs e Services de Cargo
builder.Services.AddSingleton<CargoDAO>();
builder.Services.AddSingleton<CargoService>();

//registra dependencia de DAOs e Services de Funcionario
builder.Services.AddSingleton<FuncionarioDAO>();
builder.Services.AddSingleton<FuncionarioService>();
```

AddSingleton:
Injeta automaticamente a dependência sempre que um construtor exigir as classes: CargoDAO, CargoService, FuncionarioDAO, FuncionarioService e MySqlDatabase

```
// Suporte a controllers e endpoints da API
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();

// Constrói a aplicação
WebApplication app = builder.Build();

app.UseCors("AllowAll");
// Middleware
app.UseMiddleware<ErrorHandlingMiddleware>();
app.UseStaticFiles();
app.UseAuthorization();

// Mapeia controladores
app.MapControllers();

// Mensagem no console
Console.WriteLine("🚀 API rodando em: http://localhost:8080/Login.html");

// Inicia a aplicação
app.Run();
```

Runing

- Para rodar o programa:
  - Digite no terminal:
    - dotnet run