



Checkpoint 4 - Enterprise Application Development

Introdução e conceitos:

- Model: lógica da aplicação e modelo de dados;
- View: exibe as informações para o usuário (Interface);
- Controller: recebe as requisições do usuário e encaminha o model e/ou view;

Estrutura do Projeto:

- wwwroot – diretório para arquivos estáticos, como HTML, CSS, Javascript, imagens e etc;
- Diretórios **Controllers**, **Models** e **Views** para a arquitetura MVC;
- **appsettings.json** – arquivo de configuração do projeto, como o banco de dados, por exemplo;
- Classes **Program.cs** é utilizado para configurar o comportamento da aplicação ASP.NET.

Classe Program.cs:

O arquivo **Program.cs** desempenha o papel de iniciar a aplicação. Ele define os serviços que serão utilizados e estabelece o fluxo de requisições no ASP.NET Core, composto por uma série de middleware para gerenciar solicitações HTTP.

```
using Checkpoint.Data;
using Microsoft.EntityFrameworkCore;
using CheckpointAPI.Data;

var builder = WebApplication.CreateBuilder(args);
```

```

string? connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<OracleDbContext>(options =>
{
    options.UseOracle(builder.Configuration.GetConnectionString("OracleConnection"));
});

builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

// Add services to the container.

builder.Services.AddControllers().AddNewtonsoftJson();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

```

Controller:

Em C#, um `controller` é uma classe que desempenha o papel central no padrão de design Model-View-Controller (MVC). Ele atua como um intermediário entre o modelo de dados e a interface do usuário.

Este controlador fornece endpoints para criar, listar e obter detalhes de Pokémons usando a API. Ele utiliza um contexto de banco de dados `oracleDbContext` e um mapeador `_mapper` para interagir com o banco de dados e converter entre diferentes tipos de DTOs e modelos.

```

namespace CheckpointAPI.Controllers;

[ApiController]
[Route("[controller]")]
public class PokemonController : ControllerBase
{
    private OracleDbContext _context;
    private IMapper _mapper;

    public PokemonController(OracleDbContext context, IMapper mapper)
    {
        _context = context;

```

```

        _mapper = mapper;
    }

    [HttpPost]
    public IActionResult AddPokemon([FromBody] CreatePokemonDto PokemonDto)
    {
        Pokemon pokemon = _mapper.Map<Pokemon>(PokemonDto);

        _context.Pokemons.Add(pokemon);
        _context.SaveChanges();

        return CreatedAtAction(nameof(GetPokemonById), new { id = pokemon.Id }, pokemon);
    }

    [HttpGet]
    public IEnumerable<ReadPokemonDto> GetAllUsers(
        [FromQuery] int skip = 0,
        [FromQuery] int take = 10,
        [FromQuery] string? nome = null
    )
    {
        if (nome != null)
        {
            return _mapper.Map<List<ReadPokemonDto>>(_context.Pokemons.Where((pokemon) => pokemon.Nome == nome));
        }

        return _mapper.Map<List<ReadPokemonDto>>(_context.Pokemons.Skip(skip).Take(take));
    }

    [HttpGet("{id}")]
    public IActionResult GetPokemonById(int id)
    {
        Pokemon? pokemon = _context.Pokemons.FirstOrDefault((pokemon) => pokemon.Id == id);

        if (pokemon == null)
        {
            return NotFound();
        }

        ReadPokemonDto pokeDto = _mapper.Map<ReadPokemonDto>(pokemon);

        return Ok(pokeDto);
    }

```



As configurações de rotas ficam dentro da classe Program.cs, junto as configurações padrões das rotas

Model:

Model é a representação das estruturas de dados e regras de negócio em uma aplicação. Ele encapsula a lógica de negócios e os dados associados, permitindo que a aplicação interaja com esses

dados de forma organizada.

A classe `Pokemon` serve como uma representação estruturada de um Pokémon, com propriedades que incluem informações como ID, nome, nível, experiência e tipo. Além de poder ser mapeada em uma tabela chamada "Pokemons" em um banco de dados.

```
namespace Checkpoint.Model
{
    [Table("Pokemons")]
    public class Pokemon
    {
        [Key]
        [Required]
        public int Id { get; set; }

        [Required(ErrorMessage = "Nome is required")]
        public string Nome { get; set; }
        [Required(ErrorMessage = "Nivel is required")]
        public int Nivel { get; set; }

        [Required(ErrorMessage = "Exp is required")]
        public float Exp { get; set; }
        [Required(ErrorMessage = "TipoPokemon is required")]
        public TipoPokemon Tipo { get; set; }
        public Pokemon()
        {

        }
        public Pokemon(int id, String nome, int nivel, float exp, TipoPokemon tipo)
        {
            Id = id;
            Nome = nome;
            Nivel = nivel;
            Exp = exp;
            Tipo = tipo;
        }
    }
    public enum TipoPokemon
    {
        Agua,
        Fogo,
        Planta,
        Eletrico,
        Fantasma,
        Lendario
    }
}
```

Annotation `[Required]` indica que um campo ou propriedade em uma classe é obrigatório, ou seja, não pode ser nulo ou vazio.

Annotation [Key] indicar que uma propriedade em uma classe representa a chave primária de uma entidade em um banco de dados.

ErrorMessage é usada para fornecer uma mensagem de erro personalizada quando a validação de um atributo falha.

DTOs:

Representa uma classe simples contendo dados. Ele é usado para transferir dados entre camadas de uma aplicação ou entre a aplicação e fontes externas, como um banco de dados ou uma API.

CreatePokemonDto:

Este DTO, `CreatePokemonDto`, serve para representar os dados necessários para a criação de um novo Pokémon. Ele define quatro propriedades (Nome, Nível, Experiência e Tipo) e exige que todos os campos sejam preenchidos para criar com sucesso um novo Pokémon. Se alguma dessas propriedades não for fornecida, uma mensagem de erro será gerada.

```
namespace CheckpointAPI.Data.DTOs;

public class CreatePokemonDto
{
    [Required(ErrorMessage = "Nome is required")]
    public string Nome { get; set; }

    [Required(ErrorMessage = "Nivel is required")]
    public int Nivel { get; set; }

    [Required(ErrorMessage = "Exp is required")]
    public float Exp { get; set; }

    [Required(ErrorMessage = "TipoPokemon is required")]
    public TipoPokemon Tipo { get; set; }
}
```

ReadPokemonDto

Este DTO, `ReadPokemonDto`, é utilizado para representar os dados que serão lidos ou consultados sobre um Pokémon. Ele contém informações como o ID, nome, nível, experiência e tipo do Pokémon. É uma forma eficiente de transferir apenas as informações relevantes quando se está obtendo dados sobre um Pokémon específico.

```
using Checkpoint.Model;

namespace CheckpointAPI.Data.DTOs;

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

```
    public string Nome { get; set; }
    public int Nivel { get; set; }
    public float Exp { get; set; }
    public TipoPokemon Tipo { get; set; }
}
```

UpdatePokemonDto:

Este DTO, `UpdatePokemonDto`, é utilizado para representar os dados que podem ser atualizados em um Pokémon. Ele contém informações como o nome, nível, experiência e tipo do Pokémon que podem ser alterados durante uma operação de atualização.

```
using Checkpoint.Model;
using System.ComponentModel.DataAnnotations;
namespace CheckpointAPI.Data.DTOs;
public class UpdatePokemonDto
{
    [Required(ErrorMessage = "Nome is required")]
    public string Nome { get; set; }

    [Required(ErrorMessage = "Nivel is required")]
    public int Nivel { get; set; }

    [Required(ErrorMessage = "Exp is required")]
    public float Exp { get; set; }

    [Required(ErrorMessage = "TipoPokemon is required")]
    public TipoPokemon Tipo { get; set; }
}
```

Migration:

Uma migração em C# é uma forma de **controlar e automatizar** as alterações no esquema (estrutura) de um banco de dados em um aplicativo .NET Core

Esta migração define a estrutura inicial da tabela "Pokemons" no banco de dados Oracle, incluindo suas colunas e configurações específicas do Oracle. Ao aplicar esta migração, a tabela será criada no banco de dados. Ao reverter a migração, a tabela será removida.

```
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace CheckpointAPI.Migrations
{
    /// <inheritdoc />
    public partial class Initial : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Pokemons",
```

```

        columns: table => new
    {
        Id = table.Column<int>(type: "NUMBER(10)", nullable: false)
            .Annotation("Oracle:Identity", "START WITH 1 INCREMENT BY 1"),
        Nome = table.Column<string>(type: "NVARCHAR2(2000)", nullable: false),
        Nivel = table.Column<int>(type: "NUMBER(10)", nullable: false),
        Exp = table.Column<float>(type: "BINARY_FLOAT", nullable: false),
        Tipo = table.Column<string>(type: "NVARCHAR2(2000)", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Pokemons", x => x.Id);
    });
}

/// <inheritDoc />
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Pokemons");
}
}
}

```

Esta é uma migração em C# que define a estrutura da tabela "Pokemons" em um banco de dados usando o Entity Framework Core.

- O método `Up` é chamado quando esta migração é aplicada.
 - Dentro deste método, é criada uma nova tabela chamada "Pokemons" usando o método `migrationBuilder.CreateTable`.
 - A tabela tem cinco colunas: `Id`, `Nome`, `Nivel`, `Exp` e `Tipo`.
 - As propriedades `Id`, `Nome`, `Nivel`, `Exp` e `Tipo` são definidas com seus tipos de dados e configurações específicas do banco de dados Oracle.
- A coluna `Id` é definida como uma chave primária autoincrementada usando a anotação `.Annotation("Oracle:Identity", "START WITH 1 INCREMENT BY 1")`.
- Por fim, é definida a restrição de chave primária usando `table.PrimaryKey`.

Profiles:

Refere-se a configurações específicas para mapear propriedades entre diferentes tipos de objetos usando AutoMapper.



O AutoMapper é uma biblioteca que simplifica o processo de mapeamento de dados entre objetos de diferentes tipos.

```
using AutoMapper;
using Checkpoint.Model;
using CheckpointAPI.Data.DTOs;

namespace CheckpointAPI.Profiles
{
    public class UserProfile : Profile
    {
        public UserProfile()
        {
            CreateMap<CreatePokemonDto, Pokemon>();
            CreateMap<UpdatePokemonDto, Pokemon>();
            CreateMap<Pokemon, UpdatePokemonDto>();
            CreateMap<Pokemon, ReadPokemonDto>();
        }
    }
}
```