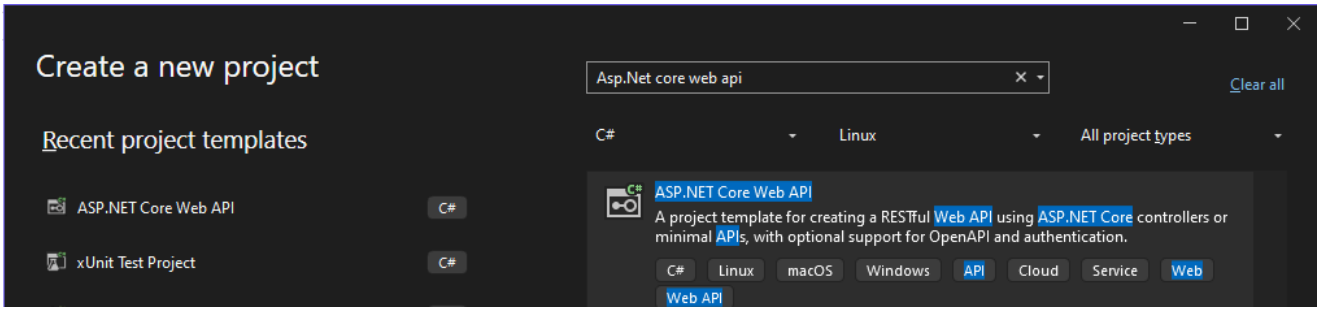
Tópicos Abordados

- Criando projeto API
- O que é MVC
- Criando Modelo de dados
- Entity Framework
- Configurar Controller com REST FULL
- Configurando Retornos customizados
- Swagger Annontation

Criando projeto API

Para criar um projeto projeto de API utilize o template > `ASP.NET Core Web API`

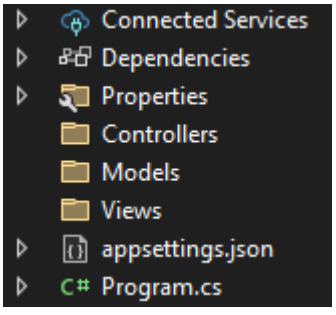


O que é MVC

ASP.NET MVC é um framework de código aberto da Microsoft para criar aplicações web. Ele usa o padrão de arquitetura **Model-View-Controller (MVC)**, que separa a aplicação em três componentes principais para facilitar a organização, a manutenção e o teste do código.

- **Controller** lida com as requisições, interage com o Model e seleciona a View correta para mostrar ao usuário;
- **Model** gerencia os dados e a lógica de negócio;
- **View** exibe a interface do usuário;

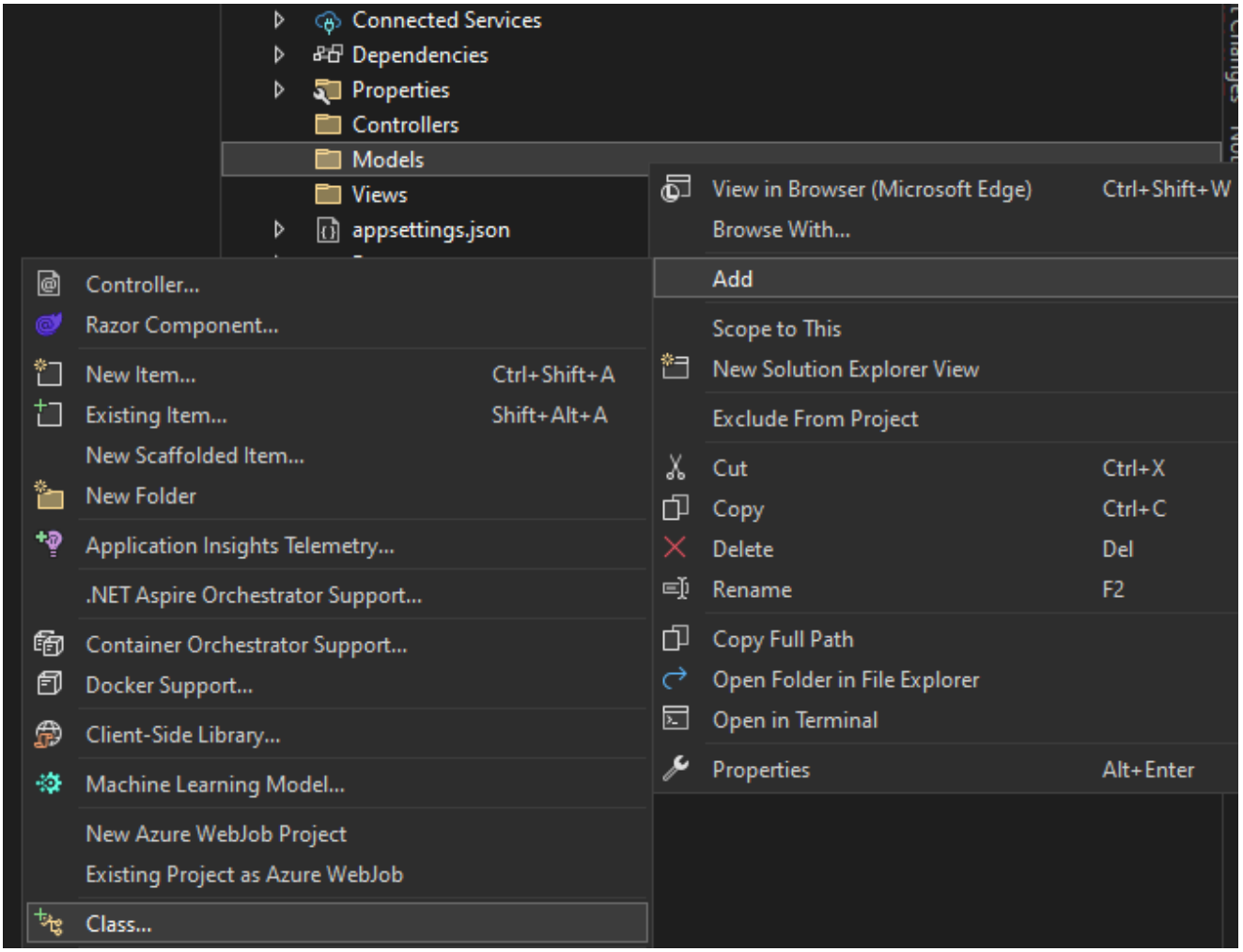
Arquitetura de pastas do projeto



Essa separação de responsabilidades permite um desenvolvimento mais estruturado e eficiente.

Criando Modelo de dados

Adicione uma classe na pasta Models para incluir um modelo de dados, clicando com botão direito do mouse na pasta models acessando o menu Add > Class...



- Cliente

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Revisao.Models
{
    [Table("tb_cliente")]
    public class ClienteEntity
    {
        [Key]
        public int Id { get; set; }

        [Required]
        [StringLength(150)]
        public string Nome { get; set; } = string.Empty;
```

```
        [StringLength(100)]
        public string Email { get; set; } = string.Empty;
        public int EstadoId { get; set; }
    }
}
```

- Estado

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Revisao.Models
{
    [Table("tb_estado")]
    public class EstadoEntity
    {
        [Key]
        public int Id { get; set; }
        public string Name { get; set; } = string.Empty;
    }
}
```

- Produtos

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Revisao.Models
{
    [Table("tb_produto")]
    public class ProdutoEntity
    {
        [Key]
        public int Id { get; set; }

        [Required]
        [StringLength(100)]
        public string Nome { get; set; } = string.Empty;
        public double Preco { get; set; }
    }
}
```

- Pedidos

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

```
namespace Revisao.Models
{
    [Table("tb_pedido")]
    public class PedidoEntity
    {
        [Key]
        public int Id { get; set; }

        [Required]
        public int Quantidade { get; set; }
    }
}
```

Entity Framework

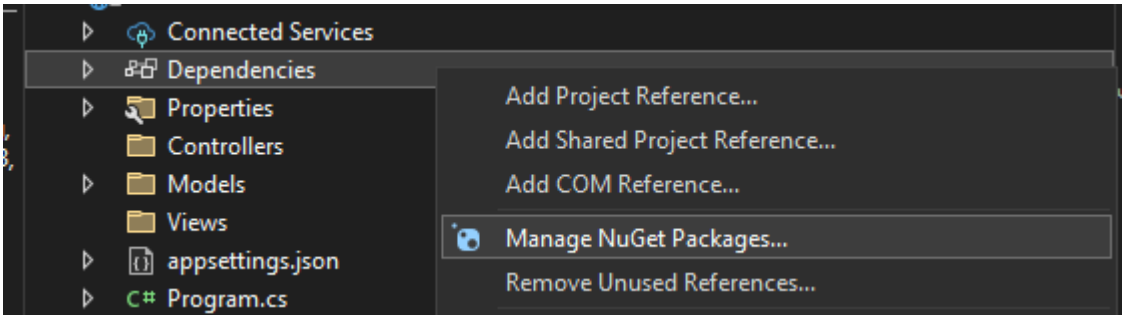
Adicionando dependencias

Incluindo as dependências do Entity Framework no projeto
Para habilitar o recurso de **gerenciamento e persistência de dados** com o **Entity Framework**, é necessário adicionar as dependências (packages) correspondentes ao seu tipo de banco de dados.

Por exemplo, para utilizar o **Entity Framework Core** com **Oracle Database**, você pode instalar os pacotes via **NuGet**:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Oracle.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Via IDE (Visual Studio)



Esses pacotes fornecem:

- **Microsoft.EntityFrameworkCore** → Núcleo do EF Core, com recursos de mapeamento objeto-relacional (ORM).
- **Oracle.EntityFrameworkCore** → Provedor para integração com bancos de dados Oracle.

- **Microsoft.EntityFrameworkCore.Tools** → Ferramentas para criar migrações, atualizar o banco e gerar código a partir de modelos.

> Após incluir as dependências, o projeto estará pronto para configurar o **DbContext** e criar as entidades que irão representar as tabelas do banco de dados.

Annontations

Ao criar modelos de dados que serão utilizados como **entidades do banco de dados**, é fundamental utilizar **annotations** (anotações) para mapear corretamente a classe e suas propriedades para a estrutura física do banco.

Essas anotações ajudam o **Entity Framework** a entender como a entidade deve ser persistida, definindo, por exemplo:

- O nome da tabela no banco de dados.
- Quais propriedades representam chaves primárias.
- Quais campos são obrigatórios.
- Restrições de tamanho e tipo de dados.

[Table] Indica o nome da tabela no banco de dados que a classe irá representar.
Se não for especificado, o Entity Framework utilizará o nome da classe como nome da tabela.

[Key] Define qual propriedade será a **chave primária** da tabela.
Por padrão, o EF reconhece propriedades chamadas `Id` ou `<NomeClasse>Id` como chave primária, mas com `[Key]` é possível especificar explicitamente.

[Required] Indica que o campo é **obrigatório** e não pode receber valor `null` no banco.

[StringLength] Define o **tamanho máximo** permitido para campos do tipo `string`.
Além de ser aplicado no banco de dados, também é usado na validação do modelo.
Quando usar: Para limitar o tamanho do texto e evitar desperdício de espaço no banco.

Configurando DbContext

O **DbContext** é o componente central do **Entity Framework Core**, responsável por:

- Gerenciar a conexão com o banco de dados.
- Mapear entidades para tabelas e propriedades para colunas.
- Executar consultas, inserções, alterações e exclusões.
- Controlar o ciclo de vida das entidades através do **Change Tracker**.

```
public class ApplicationContext : DbContext
{
    public ApplicationContext(DbContextOptions<ApplicationContext>
```

```
options)
        : base(options)
    {
    }
}
```

Configurando Injeção de Dependência

O construtor da classe **ApplicationContext** recebe um objeto `DbContextOptions<ApplicationContext>` que contém todas as configurações necessárias (string de conexão, provedor do banco, etc.). Isso permite configurar o contexto no **Program.cs** usando o método `UseOracle`, `UseSqlServer` ou outro provedor de banco de dados.
Facilita a troca do banco de dados sem alterar o código da classe.

```
using Microsoft.EntityFrameworkCore;
using Revisao.Data;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<ApplicationContext>(options => {

options.UseOracle(builder.Configuration.GetConnectionString("Oracle"));
});
```

Configurando String de conexão no appsettings.Development.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },

  "ConnectionStrings": {
    "Oracle": "Data Source=(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=
(PROTOCOL=TCP)(HOST=oracle.fiap.com.br)(PORT=1521))) (CONNECT_DATA=
(SERVER=DEDICATED)(SID=ORCL)));User Id=;Password="
  }
}
```

Configurando os Modelos no DbContext

- O `DbSet<ClienteEntity>` por exemplo, representa a tabela `Cliente` no banco.

- Cada `DbSet` é uma coleção que permite fazer consultas (`LINQ`), adicionar, atualizar ou remover registros dessa tabela.

```
public class ApplicationContext : DbContext
{
    public ApplicationContext(DbContextOptions<ApplicationContext>
options)
        : base(options)
    {
    }

    public DbSet<ClienteEntity> Cliente { get; set; }
    public DbSet<EstadoEntity> Estado { get; set; }
    public DbSet<ProdutoEntity> Produto { get; set; }
    public DbSet<PedidoEntity> Pedido { get; set; }
}
```

Migrations

Após configurar o **DbContext** e as entidades, o próximo passo é criar as **migrations** e aplicá-las ao banco de dados.
As **migrations** são um recurso do **Entity Framework Core** que permite controlar e versionar as alterações no esquema do banco de dados diretamente a partir do código.

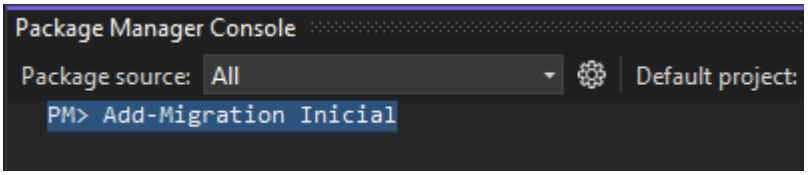
Criando as Migrations

Utilizando terminal

```
dotnet ef migrations add Inicial
```

Ou via Package Manager Console no Visual Studio

```
PM> Add-Migration Inicial
```



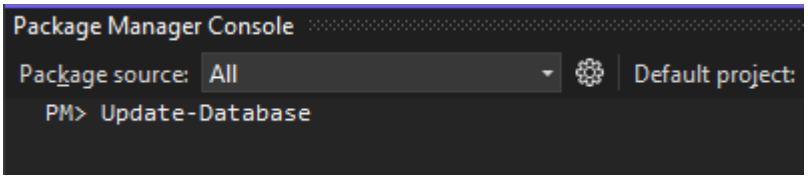
Atualizando Banco de dados

Utilizando terminal

```
dotnet ef database update
```

Ou via Package Manager Console no Visual Studio

```
PM> Update-Database
```



Configurar Controller com REST FULL

Em uma API que segue o padrão **RESTful**, uma **Controller** é a camada responsável por receber as requisições HTTP, processar (ou delegar o processamento para a camada de aplicação/serviço) e retornar a resposta adequada.

No ASP.NET Core, uma Controller RESTful:

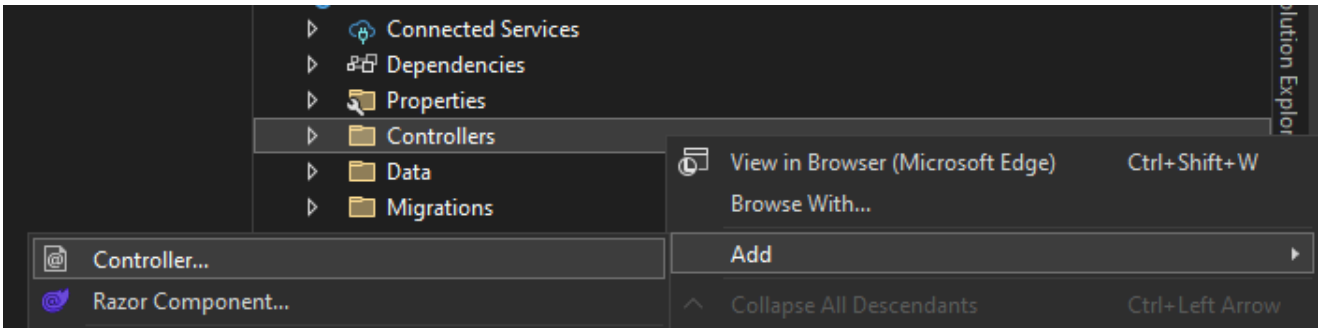
- Responde a requisições HTTP (`GET` , `POST` , `PUT` , `DELETE` etc.).
- Segue convenções de rotas para expor recursos.
- Geralmente manipula **entidades** como "Clientes", "Produtos", "Pedidos" etc.
- Não contém regras de negócio complexas, apenas **orquestra** chamadas para os serviços corretos.

Principais características de uma Controller RESTful:

1. **Recurso como foco** – cada controller representa um recurso (ex.: `ClienteController` → gerencia clientes).
2. **Uso de verbos HTTP** – para indicar a ação sobre o recurso:
   - `GET` → Buscar
   - `POST` → Criar
   - `PUT` / `PATCH` → Atualizar
   - `DELETE` → Remover
3. **URLs semânticas** – por exemplo:
   - `GET /api/clientes` → lista todos os clientes
   - `GET /api/clientes/5` → busca cliente com ID 5
   - `POST /api/clientes` → cria novo cliente
   - `PUT /api/clientes/5` → atualiza cliente com ID 5
   - `DELETE /api/clientes/5` → remove cliente com ID 5

Criando Controller

Adicionando a **Cliente** controller no projeto




```
[Route("api/[controller]")]
[ApiController]
public class ClienteController : ControllerBase
{

}
```

Injeção de dependência usando DBContext na controller

```
[Route("api/[controller]")]
[ApiController]
public class ClienteController : ControllerBase
{
    private readonly ApplicationContext _context;

    public ClienteController(ApplicationContext context)
    {
        _context = context;
    }
}
```

Metodos

- Get

```
[HttpGet]
public async Task<IActionResult> Get()
{
    var result = await _context.Cliente.ToListAsync();
    return Ok(result);
}
```

- Get(int id)

```
[HttpGet("{id}")]
public async Task<IActionResult> Get(int id)
{
```

```
    var result = await _context.Cliente.FindAsync(id);
    if (result == null)
    {
        return NotFound();
    }
    return Ok(result);
}
```

- Post

```
[HttpPost]
public async Task<IActionResult> Post(ClienteEntity entity)
{
    _context.Cliente.Add(entity);
    await _context.SaveChangesAsync();

    return Ok(entity);
}
```

- Put

```
[HttpPut("{id}")]
public async Task<IActionResult> Put(int id, ClienteEntity clienteEntity)
{
    var clienteExists = await _context.Cliente.FindAsync(id);

    if (clienteExists is not null )
    {
        clienteExists.Nome = clienteEntity.Nome;
        clienteExists.Email = clienteEntity.Email;
        clienteExists.Estado = clienteEntity.Estado;

        _context.Cliente.Update(clienteExists);
        await _context.SaveChangesAsync();

        return Ok(clienteEntity);
    }
    return NotFound();
}
```

- Delete

```
[HttpDelete("{id}")]
public async Task<IActionResult> Delete(int id)
{
    var entity = await _context.Cliente.FindAsync(id);
    if (entity is null)
```

```
    {
        return NotFound();
    }
    _context.Cliente.Remove(entity);
    await _context.SaveChangesAsync();

    return Ok(entity);
}
```

Configurando Retornos customizados

No ASP.NET Core, as Controllers retornam **códigos de status HTTP** usando métodos como `Ok()`, `NotFound()`, `NoContent()` e `BadRequest()`.
Esses métodos ajudam a seguir o padrão **RESTful** e facilitam a comunicação com o cliente da API.

1. Ok() – HTTP 200

- **Significa:** A requisição foi processada com sucesso e está retornando dados.
- **Quando usar:**
  - Ao buscar um recurso e encontrá-lo.
  - Ao executar uma operação e precisar retornar informações ao cliente.

2. NotFound() – HTTP 404

- **Significa:** O recurso solicitado **não foi encontrado** no servidor.
- **Quando usar:**
  - Quando a busca por um ID não retorna resultado.
  - Quando o recurso foi removido ou nunca existiu.

3. NoContent() – HTTP 204

- **Significa:** A operação foi bem-sucedida, mas **não há conteúdo para retornar**.
- **Quando usar:**
  - Em atualizações (`PUT`) ou exclusões (`DELETE`) que não precisam devolver dados.

4. BadRequest() – HTTP 400

- **Significa:** A requisição enviada pelo cliente **é inválida** ou contém erros de validação.
- **Quando usar:**
  - Quando parâmetros obrigatórios não foram enviados.
  - Quando o formato dos dados não é aceito.
  - Quando a validação de modelo falhar.

Tabela descrevendo o principal uso dos retornos:

Método	Código	Uso principal
Ok()	200	Retorna dados com sucesso
NotFound()	404	Recurso não encontrado
NoContent()	204	Sucesso sem retornar dados
BadRequest()	400	Erro na requisição do cliente

Swagger Annontation

- Adicionando dependencia

```
dotnet add package Swashbuckle.AspNetCore
dotnet add package Swashbuckle.AspNetCore.Annotations
dotnet add package Swashbuckle.AspNetCore.Filters
```

- Habilitando na Program.cs

```
builder.Services.AddSwaggerGen(c => {

   c.EnableAnnotations();

});
```

- Habilitando OpenDocument no projeto

```
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <!--Habilitando OpenDocument-->
        <GenerateDocumentationFile>true</GenerateDocumentationFile>
        <NoWarn>$(NoWarn);1591</NoWarn>
    <!--Habilitando OpenDocument-->
  </PropertyGroup>
```

- Descrição do metodo

```
[SwaggerOperation(

    Summary = "Lista clientes",

    Description = "Retorna a lista completa de clientes cadastrados."
```

```
)]
```

- Tipos de retornos

```
[SwaggerResponse(statusCode: 200, description: "Cliente atualizado com
sucesso", type: typeof(ClienteEntity))]

[SwaggerResponse(statusCode: 404, description: "Cliente não encontrado")]

[SwaggerResponse(statusCode: 400, description: "Requisição inválida")]
```