Desenvolvimento para *Internet* II

Criação de API Restful

Prof. Dr. Tiago Alexandre Dócusse

tad@ifsp.edu.br

Introdução

Um fluxo de funcionamento comum de um *site* pode ser representado desta forma:

Introdução

Essa arquitetura para desenvolvimento de *sites* combina todos os aspectos do sistema em um único componente (também chamado monolítico), apresentando:

Vantagens

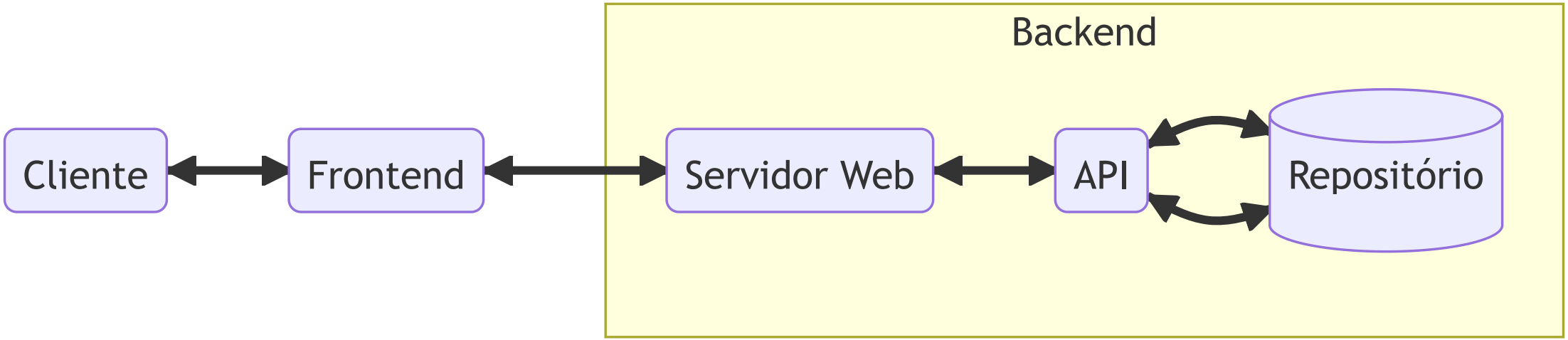
- Desenvolvimento integrado;
- Desenvolvimento rápido;
- Menos tecnologias requer menor especialização. 😕

Desvantagens

- Dificuldade no escalonamento.
- Mudança ou adição de tecnologia pode requerer muitas mudanças em alguns serviços.

Introdução

Podemos também separar o *frontend* (interface com o usuário) do *backend* (funcionamento interno do sistema), sendo que o usuário possui acesso principalmente pelo *frontend*.



Nessa arquitetura, desmembramos o sistemas em algumas partes diferentes, o que também pode ocasionar vantagens e desvantagens.

Vantagens

- Permite a utilização de uma mesma infra-estrutura para projetos de diferentes finalidades, agora ou em momento futuro;
- Permite escalonamento com diferente configuração ao *site*.

Desvantagens

- Caso não tenha nunca interesse em usar em outro projeto, desenvolvimento pode parecer não ser tão integrado;
- Necessária configuração para restrição de acesso.

Tecnologias

Diversas tecnologias podem ser utilizadas no desenvolvimento de sistemas para *internet*.

Tecnologias novas tem surgido muito rapidamente, e tecnologias antigas podem não ser eficientes para o desenvolvimento de um sistema moderno, pois:

- Utilizações são diferentes de quando a tecnologia foi projetada;
- Tecnologia (linguagens, protocolos, *software*, etc.) disponível na época;
- Expectativa dos usuários na época era diferente da atual.

Isso não quer dizer que tecnologias mais antigas não podem ser utilizadas para produzir um sistema moderno atualmente.

A substituição de uma tecnologia depende de várias perspectivas:

- Ela consegue produzir o resultado esperado em tempo e custo razoável?
- Qual a experiência da equipe com a tecnologia utilizada atualmente?
- Quão custoso (tempo/custo) seria treinar a equipe em uma tecnologia nova?
- Tecnologia nova trará ganho **efetivo** de tempo/custo?

Qual tecnologia aprender?

No entanto, ao se dispor a aprender uma tecnologia nova, ou aprender uma tecnologia pela primeira vez, é interessante aprendermos uma tecnologia moderna, a não ser que estejamos aprendendo algo para desenvolvimento de sistemas legados.

Tecnologias atuais:

- *Frontend*: HTML, CSS, Javascript, Typescript, etc.
- *Backend*: PHP, Java, Python, C#, Javascript, etc.
- *Repositório*: SQL, NoSQL, etc.

Atualmente, tornou-se comum a utilização de *frameworks* no desenvolvimento de sistemas, existindo prós e contras de sua utilização.

Vantagens na utilização de um *framework*:

- Desenvolvimento facilitado;
- Possibilidade de desenvolvimento para várias plataformas com o mesmo *framework*;
- Contratação facilitada;
- Suporte*.

Desvantagens na utilização de um *framework*

- Desempenho;
- Contratação dificultada;
- Suporte*.

Frameworks

Conjunto de ferramentas e *software* utilizados para o desenvolvimento facilitado de aplicativos (*sites*, *mobile*, etc.), contemplando várias etapas do desenvolvimento.

*Frameworks** para *frontend*

- Angular, React, Vue, Svelte, Flutter, Ionic, Next, etc.

Frameworks para *backend*

- Spring, Laravel, Cake PHP, Django, .NET, Rails, etc.

Arquitetura MVC

A arquitetura MVC é um exemplo de arquitetura, como tantas outras, que nos indica como o *software* deve/pode ser estruturado.

A utilização de uma arquitetura é essencial no sucesso da manutenção do código, e a escolha da arquitetura certa pode ser fundamental no sucesso do *software*.

Exemplo: criar um programa que leia o nome e a idade de duas pessoas, e informe a pessoa mais velha.

```
string nome1 = Console.ReadLine();
int idade1 = int.Parse(Console.ReadLine());

string nome2 = Console.ReadLine();
int idade2 = int.Parse(Console.ReadLine());

if(idade1 > idade2)
    Console.WriteLine($"{nome1} é mais velho(a)");
else if(idade2 > idade1)
    Console.WriteLine($"{nome2} é mais velho(a)");
else
    Console.WriteLine("As pessoas possuem a mesma idade");
```

O código anterior possui uma camada.

Podemos definir camada como uma parte do *software* com responsabilidades bem definidas.

Neste exemplo, a única camada existente é responsável pela entrada, processamento e saída dos dados.

Model

A *model* é uma camada responsável pela definição do contéudo dos dados.

Além disso, pode também possuir validações de como os dados devem ser armazenados.

Perceba que no exemplo anterior, precisamos criar várias variáveis pra armazenar valores de apenas 2 pessoas.

No exemplo seguinte, criaremos uma classe para representar objetos do tipo Pessoa, que possuem nome (string) e idade (int).

```
string nome = Console.ReadLine();
int idade = int.Parse(Console.ReadLine());
Pessoa p1 = new Pessoa(nome, idade);

nome = Console.ReadLine();
idade = int.Parse(Console.ReadLine());
Pessoa p2 = new Pessoa(nome, idade);

if(p1.Idade > p2.Idade)
    Console.WriteLine($"{p1.Nome} é mais velho(a)");
else if(p2.Idade > p1.Idade)
    Console.WriteLine($"{p2.Nome} é mais velho(a)");
else
    Console.WriteLine("As pessoas possuem a mesma idade");

public class Pessoa {
    public string Nome {get;set;}
    public int Idade {get;set;}

    public Pessoa(string nome, int idade) {
        Nome = nome;
        Idade = idade;
    }
}
```

Controller

A camada *controller* é a responsável pela lógica de negócio do sistema.

A utilizamos para encapsular métodos e ações a serem executadas.

Quando a interface deseja realizar uma ação (que não seja de interface), ela deve chamar uma ação na camada *controller*.

Também pode ser utilizada para validações de segurança e operações com um repositório de dados*.

View

A camada *view* é a responsável pela interface de usuário.

Deve receber programação apenas referente à interface.

Programação de lógica de negócio, validação, etc., devem ser feitas na *controller*.

No exemplo anterior, a entrada de dados é responsabilidade da camada de visão (View).

Feita a entrada de dados, a *view* fornece à *controller* os dados necessários para o cálculo.

A *controller* retorna o resultado processado para a *view*.

A *view* exibe o resultado de forma adequada.

```
string nome = Console.ReadLine();
int idade = int.Parse(Console.ReadLine());
Pessoa p1 = new Pessoa(nome, idade);

nome = Console.ReadLine();
idade = int.Parse(Console.ReadLine());
Pessoa p2 = new Pessoa(nome, idade);

Pessoa? resultado = Controller.Maior(p1, p2);

if(resultado != null)
    Console.WriteLine($"{resultado.Nome} é mais velho(a)");
else
    Console.WriteLine("As pessoas possuem a mesma idade");

public static class Controller{
    public static Pessoa? Maior(Pessoa p1, Pessoa p2){
        if(p1.Idade > p2.Idade)
            return p1;
        else if(p2.Idade > p1.Idade)
            return p2;
        return null;
    }
}

public class Pessoa {
    public string Nome {get;set;}
    public int Idade {get;set;}

    public Pessoa(string nome, int idade) {
        Nome = nome;
        Idade = idade;
    }
}
```

Perceba que utilizamos a interface para receber os dados.

Umas vez com os dados preenchidos, chamamos a classe `Controller` para realizar o cálculo.

De posse do resultado, fazemos a exibição do resultado na próprio interface.

Vantagens

- Separação clara das responsabilidades;
- Facilidade de manutenção;
- Possibilidade de utilização de diferentes interfaces e de diferentes tipos, caso o restante do sistema esteja em um *web service*.

Desvantagens

- Maior quantidade de código;
- Programadores inexperientes podem pular chamadas entre camadas;
- Código mais lento*.

API e *backend*

Como visto anteriormente, podemos utilizar diferentes arquiteturas para criar um sistema *web*.

No entanto, vamos criar nosso *site* com três componentes principais: *site*, Application Programming Interface (API) e repositório.

Frontend

- *Site* (reponsável pela exibição e interação com o usuário);

Backend

- API (responsável pela execução das regras de negócio);
- Repositório (responsável pelo armazenamento dos dados).

API e *backend*

Criaremos a API como um *webservice* com as camadas Controller e Model, de forma que ela pode ser consumida por qualquer serviço que conheça seu endereço e tenha acesso a ela.

O *site* será criado em React/Next, sendo utilizado como a interface com o usuário (camada View).

Este *site* fará requisições à API que criaremos, informando os dados de entrada e recebendo o resultado.

API RESTful

Uma API RESTful é uma API em conformidade com a arquitetura *Representational State Transfer* (REST).

Surgiu como uma alternativa para a arquitetura *Simple Object Access Protocol* (SOAP).

Características (1/2)

- Arquitetura cliente/servidor com solicitações HTTP;
- Não armazena dados entre solicitações;
- Cada solicitação deve ser suficiente para sua execução;
- Armazenamento de dados em *cache* para otimizar transações;

Requisição

Características (2/2)

- Interface uniforme entre os componentes;
- Utilização de camadas para separação de serviços; MVC
- Dados acessados por *Uniform Resource Identifier* (URI);
- Utilização de requisição específica de acordo com a ação a ser realizada.

Vamos criar nossa API em .NET.

Motivos

- Código livre e gratuito;
- Executável em Windows, Mac e Linux.

Versão atual do .NET framework: 9.0 (família `core`), mas usaremos a versão 8.0.

Necessária instalação do SDK na máquina que for executar.

Utilizaremos o VSCode para programação (pode ser usado qualquer editor de textos).

API mínima vs API com controlador

O .NET suporta dois tipos de projetos de API: mínima e com controlador.

API mínima

- Código mais enxuto;
- Não há necessidade de criar classes para *controllers*;
- Indicada para situações com uma ou poucas *models*.

API com controlador

- Código maior;
- Separação de responsabilidades por controlador;
- Alguns recursos a mais.

Criando a API

Abra um programa que permita a digitação de comandos (prompt, console, powershell, etc.), e digite o seguinte código:

```
dotnet new webapi --use-controllers -o apiExemplo -f net8.0
```

`apiExemplo` é o nome do projeto que criamos, podendo ser qualquer nome válido.

Caso a opção `-f net8.0` seja omitida, será utilizada a versão do *framework* mais recente disponível no PC onde o projeto está sendo criado.

Para abrir o VSCode com o projeto criado:

```
code -r apiExemplo
```

Estrutura criada

No projeto criado, os principais arquivos e pastas são os seguintes:

`Controllers` pasta onde devem ficar as *controllers*;
`appsettings.json` arquivo com configurações do projeto;
`Program.cs` arquivo inicial da API;
`WeatherForecast` exemplo de uma *model*.

Para compilar o código, no terminal digite:

```
dotnet build
```

Para executar (após a compilação ter sido realizada com sucesso), no terminal, digite:

```
dotnet run
```

Do jeito feito anteriormente, para que uma alteração no código possa ser feita, devemos encerrar o processo e fazer a compilação e a execução novamente.

Podemos utilizar o seguinte comando para que, caso alguma alteração no código ocorre, o *framework* automaticamente faça a compilação e a execução:

```
dotnet watch
```

Caso executemos este projeto inicial, devemos visitar a rota `/weatherForecast` para visualizar um exemplo de saída da API.

É possível acessar a rota `/swagger` para utilizar o Swagger e verificar a API.

Do jeito que está configurado o projeto, o Swagger só será ativado em modo depuração.

Preparando o projeto

Para criar o nosso projeto, podemos fazer algumas alterações:

- Apague o arquivo `/WeatherForecast.cs`;
- Apague o arquivo `Controllers/WeatherForecastController.cs`;
- Crie a pasta `/Models`. Ela conterá todas nossos arquivos de Model;
- Crie a pasta `/Data`. Ela conterá os arquivos de conexão com o repositório de dados.

Iremos usar inicialmemte como exemplo o cadastro de um tipo de curso de uma instituição, que possui os seguintes dados:

Nome	Tipo	Restrições
Id	Inteiro	Chave primária. Obrigatório
Nome	Texto	Mínimo três caracteres. Máximo cem caracteres. Obrigatório
Descrição	Texto	Mínimo cinco caracteres. Máximo cem caracteres. Obrigatório

Crie o arquivo `/Models/TipoCurso.cs`, e dentro dele, crie uma classe com os dados referentes a um tipo de curso, como no código a seguir:

```
public class TipoCurso
{
    public int Id { get; set; }
    public string Nome { get; set; }
    public string Descricao { get; set; }
}
```

O nome da *model* deve ser apenas um identificador válido, mas é interessante manter o padrão por todo o projeto (TipoCurso, TipoCursoModel, MTipoCurso, etc.)

Atributos para propriedades das *models*

É possível utilizar alguns atributos nas propriedades das *models*, de forma que seja feita uma validação automática pelo *framework* no momento do recebimento da requisição.

O atributo mais comum é o `Required`, que está declarado em `System.ComponentModel.DataAnnotations`.

Utilize `[Required]` antes de cada propriedade para informar que aquela propriedade deve ter um valor no recebimento da requisição.

```
using System.ComponentModel.DataAnnotations;

public class TipoCurso
{
    [Required]
    public int Id { get; set; }

    [Required]
    public string Nome { get; set; }

    [Required]
    public string Descricao { get; set; }
}
```

Caso o atributo não seja respeitado em uma requisição, o próprio *framework* retornará uma mensagem de erro.

Caso queiramos personalizar esta mensagem, podemos usar o argumento `ErrorMessage` com o atributo `Required`. Caso não o utilizemos, a mensagem padrão do *framework* será exibida.

```
using System.ComponentModel.DataAnnotations;

public class TipoCurso
{
    [Required]
    public int Id { get; set; }

    [Required(ErrorMessage = "O nome é obrigatório")]
    public string Nome { get; set; }

    [Required]
    public string Descricao { get; set; }
}
```

Alguns outros atributos podem ser utilizados:

Nome	Descrição
Range	Define intervalos para valores inteiros ou DateTime
MinLength	Define o comprimento mínimo de uma *string*
MaxLength	Define o comprimento máximo de uma *string*
RegularExpression	Define uma expressão regular para validar a propriedade
EmailAddress	Define que o campo é uma string do tipo e-mail

É possível utilizar mais de um atributo por propriedade.

Estes atributos também permitem a utilização de uma mensagem de erro personalizada.

```
using System.ComponentModel.DataAnnotations;

public class TipoCurso
{
    [Required]
    public int Id { get; set; }

    [Required(ErrorMessage = "O nome é obrigatório")]
    [MinLength(3)]
    [MaxLength(100, ErrorMessage = "O nome deve possuir, no máximo, 100 caracteres")]
    public string Nome { get; set; }

    [Required]
    [MinLength(5, ErrorMessage = "A descrição deve possuir, no mínimo, 5 caracteres")]
    [MaxLength(100)]
    public string Descricao { get; set; }
}
```

Outros exemplos:

```
using System.ComponentModel.DataAnnotations;

public class Exemplo
{
    [Required]
    public int Id { get; set; }

    [Required]
    [Range(2, 12)]
    public int Semestres { get; set; }

    [Required]
    [RegularExpression(@"(\b25[0-5]|\b2[0-4][0-9]|\b[01]?[0-9][0-9]?)(\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)){3}")]
    public string IPv4 { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [Range(typeof(DateTime), "01-12-2024", "31-12-2024")]
    public DateTime Data { get; set; }
}
```

Definida a *model*, vamos criar a *controller* em `/Controllers`.

Crie o arquivo `/Controllers/TipoCursoController.cs`.

A controller poderá ser acessada através de uma rota, do tipo `https://endereçodaapi/nomedarota`;

Não é necessário usar o `Controller` no final do nome, mas por padrão, o *framework* remove o `Controller` da rota da Controller, ficando a rota `/TipoCurso`.

É importante manter um padrão de nomenclatura para todas as Controllers.

Normalmente a rota será no formato: `/controller/action/id`, sendo:

- `controller`: a Controller a ser acessada;
- `action`: a ação da Controller a ser executada;
- `id`: o *id* da Model sobre a qual a ação deve ser executada.

Para Controllers acessadas pela rota "/api", crie o arquivo `/Controllers/TipoCursoController.cs` com o código a seguir:

```
using Microsoft.AspNetCore.Mvc;

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

}
```

É necessário fazer a herança da classe base `ControllerBase`.

Ações comuns na *controller*

Ações na *controller* são métodos que podem ser acessados através da API.

É possível diferenciar as ações pelo nome ou pelo método da requisição.

Também é possível definir uma rota para a ação diferente do seu nome.

Um método que não deve ser acessado como uma ação na API deve possuir o atributo `[NonAction]` antes de sua declaração.

Nome	Verbo HTTP	Argumento(s)	Descrição
Get	Get		Retorna todas os *models* da rota
Get	Get	id	Retorna a *model* referente ao id informado
Post	Post	objeto	Armazena o objeto informado
Put	Put	id, objeto	Atualiza o objeto informado
Delete	Delete	id	Remove o objeto referente ao id informado

Para testarmos a API, crie este método de exemplo dentro de `TipoCursoController`:

```
[HttpGet]
public string Get()
{
    return "Olá mundo";
}
```

É possível navegar para a rota `/api/TipoCurso` no navegador para verificar o seu funcionamento, ou então, verificar seus dados através do Swagger.

Também podemos utilizar o Postman para fazer uma requisição GET para o mesmo endereço.

Caso apresente erro de SSL, desabilite a verificação de certificados no Postman.

O retorno de uma ação na API pode ser de qualquer tipo, como *string* utilizada no exemplo anterior.

Vamos utilizar o retorno como um objeto do tipo `ActionResult`, que nos permite retornar objetos de diferentes tipos, bem como objetos que representam códigos de *status* HTTP, como `Ok`, `BadRequest`, `NotFound`, etc., dependendo da necessidade.

Dessa forma, podemos alterar o código anterior para:

```
[HttpGet]
public ActionResult Get()
{
    return Ok("Olá mundo");
}
```

Pode ser necessário parar a execução e iniciar novamente.

Nem sempre o retorno desejado de uma ação da API é uma string. Ela pode ser uma Model ou uma coleção de Models, por exemplo.

Quando o retorno de uma ação na API for uma coleção de objetos, devemos utilizar `ActionResult<IEnumerable<NomeModel>>` como tipo de retorno. Por exemplo, substitua o método anterior para:

```
[HttpGet]
public ActionResult<IEnumerable<TipoCurso>> Get()
{
    return new List<TipoCurso>();
}
```

Esta é a ação que, futuramente, irá buscar todas as Models de `TipoCurso` no repositório.

A ação para inserir uma Model no repositório é a ação `Post`.

Para testar a validação da *model*, vamos criar uma ação do tipo POST, que recebe um objeto da *model* criada:

```
[HttpPost]
public ActionResult Post(TipoCurso item)
{
    return Ok("Apenas validando os dados");
}
```

Para fazer uma requisição a esta ação no Postman, mude o tipo da requisição para `Post`, adicione ao corpo (*body*) da requisição um objeto do tipo JSON sem atributos, e veja que a requisição só é executada com sucesso quando os atributos estão de acordo com o especificado na *model* que fizemos anteriormente.

Exemplo de objeto para requisição com erros:

```
{

}
```

Exemplo de objeto para requisição sem erros:

```
{
    "Nome": "Superior",
    "Descricao": "Cursos superiores de educação"
}
```

Lembre-se que todas as validações informadas na Model são realizadas, como tamanho mínimo e máximo de caracteres.

Repositório de dados

Vamos utilizar um repositório de dados para armazenar os dados de forma permanente.

A forma de armazenamento é dependende do projeto: arquivo, banco de dados relacional, banco de dados não relacional, etc.

Podemos fazer o acesso aos dados diretamente na ação da *controller*.

Vantagem

- menor código

Desvantagem

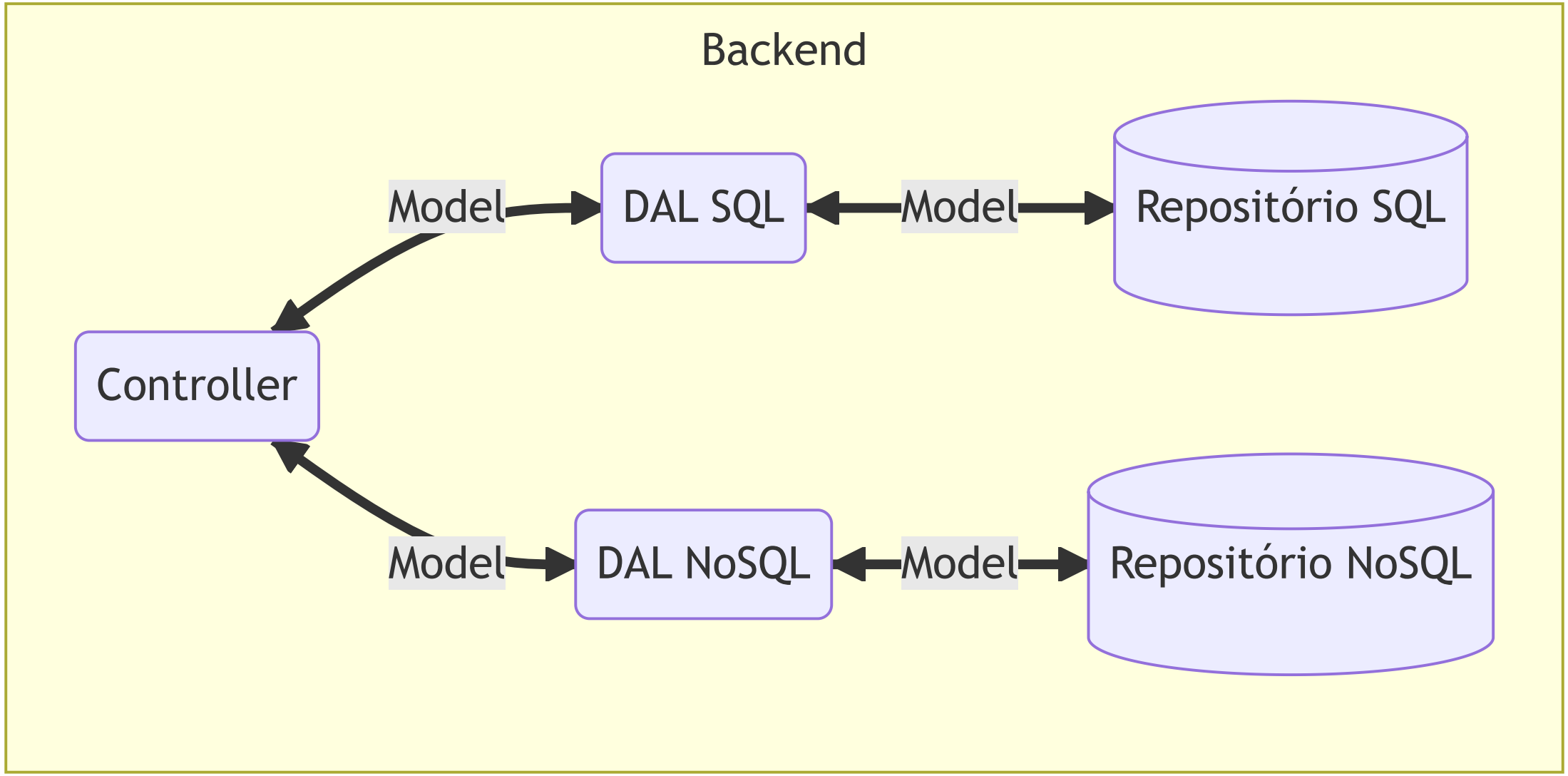
- caso desejamos trocar a forma de armazenamento, precisamos reprogramar parte da *controller*

A utilização de uma camada de dados - *Data Access Layer* (DAL) - entre a *controller* e o repositório pode amenizar esta desvantagem.

Além disso, caso tenhamos diferentes tipos de repositório, podemos ter uma camada de dados pra cada repositório, isolando a *controller* de detalhes técnicos do repositório.

A programação da lógica de negócio ainda deve ficar na *controller*. Na camada de dados devem ficar apenas os códigos relacionados ao acesso ao repositório.

Neste exemplo, não vamos usar a camada de dados, pois vamos trabalhar apenas com um repositório.


Backend
Model
DAL SQL
Model
Repositório SQL
Controller
Model
DAL NoSQL
Model
Repositório NoSQL

Neste exemplo, vamos utilizar o Microsoft SQL Server como repositório de dados relacional deste projeto.

Vantagens

- Gratuito (versão community);
- Executável no Windows, Linux ou *container* Docker.

Desvantagens

- Dimensionamento.

Execução do SQL Server

Podemos executar o SQL Server de algumas formas diferentes:

- Localmente de forma nativa;
- Localmente em um *container*;
- Remoto (rede ou nuvem) de forma nativa;
- Remoto (rede ou nuvem) em um *container*.

Para execução nativa, faça o *download* e instale com as configurações desejadas:
https://www.microsoft.com/pt-br/sql-server/sql-server-downloads

Para execução na rede, basta instalar em uma máquina na rede, ou usar um serviço na nuvem para ter um servidor rodando com um endereço IP válido.

Para execução em um *container* no Docker, inicie o Docker e baixe a imagem do SQLServer:

```
docker pull mcr.microsoft.com/mssql/server:2022-latest
```

Em seguida, inicie um *container* com as configurações desejadas:

```
docker run --name Banco -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=Senh@123" -e "MSSQL_PID=Express" -p 1433:1433 -v data:/var/opt/mssql -d mcr.microsoft.com/mssql/server:2022-latest
```

Algumas opções que utilizamos no comando anterior são:

- `--name Banco` Define `Banco` como o nome do *container*;
- `-e ACCEPT_EULA=Y` Informa que estamos de acordo com o contrato de licença do usuário final do MSSQL;
- `e "MSSQL_SA_PASSWORD=Senh@123"` Define a senha do super usuário do SGBD;
- `-e "MSSQL_PID=Express"` Informa que usaremos a versão gratuita do SGBD;
- `-p 1433:1433` Mapeia a entrada de conexões da porta 1433 para a porta 1433;
- `-v data:/var/opt/mssql` Armazena os dados do MSSQL em um volume;
- `-d` Inicia o *container* como um processo no *background*.

Podemos acessar o repositório com o SQL Server Management Studio ou o Azure Data Studio para verificar que ele está no ar e funcionando corretamente.

Para isso, utilize os seguinte parâmetros de conexão:

- `Endereço` localhost, 1433
- `Usuário` sa
- `Senha` Senh@123 (ou a senha utilizada na criação do *container*)

Comunicação com o repositório

Para comunicar com o repositório, podemos criar o código das consultas e executá-las quando necessário, com a utilização de bibliotecas próprias para isso, como o ADO.NET.

Também podemos utilizar um *Object-Relational Mapper* (ORM) - Mapeador Objeto Relacional - para facilitar a execução das consultas, especialmente as mais simples e corriqueiras.

Vamos usar o Entity Framework Core como ORM.

Para consulta de repositórios que funcionam com o Entrity Framework, acesse https://learn.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli.

Vantagens de se utilizar um ORM

- Facilidade de execução de consultas simples
- Criação "automática" do repositório
- Fácil troca de tecnologia do repositório

Desvantagens

- Complexidade para consultas complexas
- Tipo de repositório não suportado
- Para ORMs de terceiros, suporte e adição de funcionalidades limitados

Entity Framework Core

Para instalar o Entity Framework no projeto, digite o seguinte comando:

```
dotnet add package Microsoft.EntityFrameworkCore
```

Atente-se à versão do .NET utilizado, pois o comando acima irá instalar a versão mais recente do pacote informado. O ideal é instalar a versão comum à versão do .NET instalado.

Caso queira fazer uma instalação com uma versão diferente da mais recente, a informe no comando:

```
dotnet add package Microsoft.EntityFrameworkCore –v 8.0.3
```

É necessário instalar um provedor de serviços para que o Entity Framework consiga trabalhar com o repositório utilizado (no caso deste exemplo o SQL Server). Para isso, digite o seguinte comando:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Opcionalmente, instale o pacote de ferramentas do Entity Framework para gerar atualizações na base de dados diretamente do projeto, sem a necessidade de executar código SQL.

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

Camada de Dados

Não vamos utilizar uma camada de dados neste exemplo para facilitar o desenvolvimento.

No entanto, criaremos a classe para trabalhar com o Entity Framework na pasta `/Data`, pois é o local onde faríamos as classes da camada de dados.

Crie uma classe pública/interna em '/Data' que herda de `Microsoft.EntityFrameworkCore.DbContext`.

Crie um construtor para esta classe, com `DbContextOptions<DataContext> options` como argumento, passando este objeto para o construtor da classe base.

```
using Microsoft.EntityFrameworkCore;

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

Nesta classe, para cada *model* mapeada a uma tabela, devemos criar uma propriedade `DBSet<T>`, em que o tipo abstrato é a *model* desejada. No nosso exemplo:

```
public DbSet<TipoCurso> TipoCursos { get; set; } = null!;
```

Veremos como fazer modificações nas tabelas na próxima aula.

Comunicando com o repositório

Agora precisamos informar o local onde o repositório está localizado.

É comum armazenar esta informação no arquivo de configurações do projeto, localizado em `/appsettings.json`.

Adicione um atributo colecionável `ConnectionStrings` ao objeto representado em `appsettings.json`:

```
"ConnectionStrings": {
    "Development": ""
  }
```

A *string* de conexão irá se alterar, de acordo com a forma que a conexão será feita.

Alguns atributos comuns são:

- `Server` : o endereço do servidor
- `Database` : o nome da base de dados
- `Trusted_Connection` : *true*, caso for autenticar no repositório com o usuário autenticado no sistema operacional
- `User` : o nome do usuário para autenticar no repositório
- `Password` : a senha do usuário para autenticar no repositório

Exemplos

- Conexão na máquina local com usuário autenticado pelo sistema operacional:

```
"Server=localhost; Database=NomeBase; Trusted_Connection=True"
```

- Conexão na máquina local com usuário e senha definidos:

```
"Server=localhost; Database=NomeBase; User=sa; Password=Senh@123"
```

- Conexão em máquina na rede:

```
"Server=192.168.10.16; Database=NomeBase; Trusted_Connection=True"
```

- Conexão em máquina local com instância nomeada:

```
"Server=192.168.1.1\\NomeInstancia; Database=NomeBase; Trusted_Connection=True"
```

- Conexão em um *container* do Docker em porta específica, com a API fora de um *container*:

```
"Server=localhost,1433; Database=NomeBase; User=sa;Password=Senh@123"
```

- Conexão em *container* do Docker, com a API dentro de um *container*:

```
"Server=container_repos; Database=NomeBase; User=sa;Password=Senh@123"
```

O arquivo `appsettings.json` ficará parecido com o seguinte:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DevelopmentPath": "Server=localhost,1433; Database=NomeBase; User=sa;Password=Senh@123"
  }
}
```

Por fim, precisamos informar ao projeto que ele deverá se comunicar com o repositório.

Em `Program.cs`, após a linha em que informa-se que o serviço irá usar *controllers*, digite o código:

```
builder.Services.AddDbContext<DataContext>(options =>
    options.UseSqlServer("name=ConnectionStrings:Development"));
```

Perceba que:

- Utilizamos a classe criada na pasta `/Data` como tipo abstrato;
- Utilizamos o nome da *string* de conexão criada anteriormente.

Pode ser necessário adicionar a seguinte linha no início do arquivo:

```
using Microsoft.EntityFrameworkCore;
```

Geração e manutenção da base de dados

O projeto já se encontra configurado para conectar ao repositório.

No entanto, ainda falta criar as tabelas onde os dados serão armazenados, bem como seus relacionamentos e restrições.

Podemos criá-las manualmente executando seu código SQL no SQL Management Studio ou Azure Data Studio.

Ou podemos utilizar o pacote de ferramentas do Entity Framework que instalamos anteriormente para gerar o código.

Caso não esteja instalado um utilitário do dotnet, é necessário digitar o comando:

```
dotnet tool install --global dotnet-ef
```

Caso ele esteja instalado em uma versão anterior, atualize-o, com o comando:

```
dotnet tool update --global dotnet-ef
```

Sempre que houver alteração em alguma *model* que seja mapeada ao repositório, ou no repositório em si, devemos criar uma nova **migração**.

Para criar uma migração, digite o seguinte comando:

```
dotnet ef migrations add NomeMigracao
```

O projeto será compilado e, em caso de sucesso, a nova migração será criada.

Para atualizar a base de dados com base na última migração criada, digite o seguinte comando:

```
dotnet ef database update
```

Caso você veja um erro de certificado, adicione este treho à *string* de conexão:

```
TrustServerCertificate=True;
```

Para atualizar a base de dados para uma migração específica, utilize o comando:

```
dotnet ef database update NomeMigracao
```

Caso deseje remover a última migração (antes de ter atualizado a base), digite o comando:

```
dotnet ef migrations remove
```

Para reverter a base de dados para um estado inicial, utilize o comando:

```
dotnet ef database update 0
```

Configurando a *controller* para acessar o repositório

Em `TipoCursoController`, crie um objeto privado e somente leitura da classe criada em `/Data`.

Ele deve receber o valor passado como argumento no construtor desta classe.

```
[Route("api/[controller]")]
[ApiController]
public class TipoCursoController : ControllerBase
{
    private readonly DataContext context;

    public TipoCursoController(DataContext _context)
    {
        context = _context;
    }
}
```

Ação de inserção na API

Criada a base de dados, vamos fazer uma ação de inserção de tipo de curso na API.

Na próxima aula veremos mais detalhes do Entity Framework.

Nossa *model* continuará a mesma que fizemos anteriormente:

```
using System.ComponentModel.DataAnnotations;

public class TipoCurso
{
    [Required]
    public int Id { get; set; }

    [Required(ErrorMessage = "O nome é obrigatório")]
    [MinLength(3)]
    [MaxLength(100, ErrorMessage = "O nome deve possuir, no máximo, 100 caracteres")]
    public string Nome { get; set; }

    [Required]
    [MinLength(5, ErrorMessage = "A descrição deve possuir, no mínimo, 5 caracteres")]
    [MaxLength(100)]
    public string Descricao { get; set; }
}
```

Em `TipoCursoController`, crie um método Post para salvar um tipo de curso no repositório. Seu retorno deve ser um `ActionResult`.

Utilize `context.NomeTabela.Save` para informar qual objeto deve ser salvo.

Utilize `context.SaveChanges()` para salvar as alterações na base de dados.

Retorne `Ok` em caso de sucesso ou `BadRequest` em caso de erro.

Opcionalmente, retorne uma mensagem adequada de acordo com o resultado obtido.

```
[HttpPost]
public ActionResult Post(TipoCurso item)
{
    try
    {
        context.TipoCursos.Add(item);
        context.SaveChanges();
        return Ok("Tipo de curso salvo com sucesso");
    }
    catch
    {
        return BadRequest("Erro ao salvar o tipo de curso informado");
    }
}
```

Execute o Postman ou, pelo Swagger, faça uma requisição POST para `/api/TipoCurso` com o seguinte objeto JSON no corpo da requisição:

```
{
    "Nome": "Superior",
    "Descricao": "Cursos superiores de tecnologia"
}
```

Este curso deve ser salvo no repositório.

Não é necessário informar o Id do curso, pois ele é um campo identidade/auto-incrementável. Veremos mais detalhes sobre isso na próxima aula.

Ação de listagem na API

Para listar todos dados da tabela TipoCurso, crie uma nova ação GET em `TipoCursoController`, com retorno `ActionResult<IEnumerable<TipoCurso>>`, sem argumentos.

Utilize `context.TipoCursos.ToList()` para retornar a lista desejada.

```
[HttpGet]
public ActionResult<IEnumerable<TipoCurso>> Get()
{
    try
    {
        return Ok(context.TipoCursos.ToList());
    }
    catch
    {
        return BadRequest("Erro ao listar os tipos de curso");
    }
}
```

Acesse a API em `/api/TipoCurso` para ver a listagem de tipos de curso no navegador, no Postman, ou use o Swagger para verificar se os dados foram armazenados de forma correta.