**Aula 3 - Resumo 1:** Já temos a aplicação ASP.NET Core MVC rodando, porém, por enquanto temos somente as views, ou seja, as páginas com conteúdo estático em HTML. O próximo passo será passar os dados para estas views, para podermos formar uma página de verdade.

Teremos que criar um modelo, que fornecerá estes dados para o MVC. Para isso, serão necessárias determinadas classes. Normalmente, é necessária a criação de um banco de dados, que nos permita acessar estes dados. Contudo, faremos uma abordagem diferente, chamada *code first*, ou código primeiro. Isso significa que criaremos as classes e, em seguida, criaremos o banco de dados com base nestas.

As classes são representadas por quatro entidades, que aqui são: Pedido, Cadastro, ItemPedido e Produto. No Pedido, temos os itens do carrinho, e associado a ele temos o Cadastro, com os dados do usuário. Temos também o ItemPedido, com as peculiaridades do pedido, como quantidade e preço e, por fim, temos o Produto, que são os itens presentes no catálogo, no carrossel.

Cada uma das classes possui um atributo, uma propriedade, chamada id, que serve como chave primária para esta entidade. Temos ainda, uma propriedade de navegação, responsáveis por realizar a associação entre uma classe e outra. Por exemplo, temos o Pedido relacionado a Cadastro. Ou seja, é possível acessar o Cadastro a partir de uma classe Pedido, com os dados do usuário.

Retornaremos ao Visual Studio, e abriremos a pasta "\_Recursos > dados".

Nela, encontramos um arquivo chamado modelo.cs. Nele, há diversos exemplos de classes, que são justamente aqueles abordados acima. Copiaremos este arquivo e colaremos na pasta "CasaDoCodigo > Models".

Contudo, o modelo, sozinho, é incapaz de fazer algo. Precisamos fazer ainda uma associação com o banco de dados. Para que possamos gerar tabelas a partir do nosso código, precisamos de um intermediário, que chamamos de **Entity Framework Core**. Ele faz o mapeamento do objeto relacional, e gerará a partir das classes as tabelas, campos e chaves necessários no banco de dados. Posteriormente, ele gerenciará esta troca de informações entre o banco e o sistema. Ele é fundamental para fazermos o acesso aos dados.

Assim, temos as classes do modelo, e precisamos agora criar um contexto para o Entity Framework Core saber como lidar com as entidades para fazer o mapeamento. Para isso, clicaremos com o botão direito do mouse sobre a pasta "CasaDoCodigo" e selecionaremos a opção "Add > Class...". A chamaremos de ApplicationContext.

Esta nova classe precisa herdar de DbContext, lembrando de importar o using. Precisamos inserir ainda um construtor, com o parâmetro options. Clicaremos com o botão direito do mouse sobre Generate constructor 'ApplicationContext(options)'. Removeremos o atributo NotNullAtribute pois ele não será necessário. Criaremos também um override para substituirmos um método, e assim criarmos o modelo.

**namespace CasaDoCodigo**

**{**

**public class ApplicationContext : DbContext**

**{**

**public ApplicationContext(DbContextOptions options) : base(options)**

**{**

**}**

**protected override void OnModelCreating(ModelBuilder modelBuilder)**

**{**

**base.OnModelCreating(modelBuilder);**

**}**

**}**

**}**

Quando estiver criando o modelo, ele acessará este método para poder fazer o mapeamento. Dando continuidade, faremos o mapeamento a partir do parâmetro modelBuilder e chamaremos o método Entity, que registrará uma classe do nosso modelo, no caso Produto, e o mapearemos para que possa ser adicionado ao modelo. Caso contrário, o Entity Framework não saberá que classes utilizará para gerar o banco de dados.

Para registrar que o método contém uma chave primária, chamaremos o método HasKey e o nome da propriedade que representa a referida chave, ou seja, (t => t.Id):

namespace CasaDoCodigo

{

public class ApplicationContext : DbContext

{

public ApplicationContext(DbContextOptions options) : base(options)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<Produto>().HasKey(t => t.Id);

}

}

}

***Você acabou de criar um modelo de dados utilizando código C#. Agora, esse modelo será usado para mapear tabelas de um banco de dados relacional SQL Server, utilizando o framework Entity Framework Core. Qual componente é necessário desenvolver para fazer o Entity Framework Core realizar operações de consultas e alterações no banco de dados utilizando as entidades que você acabou de criar?***

***Uma classe que implementa a classe-base DbContext.*** DbContext é uma classe do namespace Microsoft.EntityFrameworkCore. Uma instância DbContext representa uma sessão com o banco de dados e pode ser usada para consultar e salvar instâncias das suas entidades. O DbContext é uma combinação dos padrões Unit Of Work e Repository.

**Resumo 2:** Nesta aula, criaremos um banco de dados vazio para podermos fazer a migração das classes e gerar as tabelas do banco de dados. Clicaremos no menu "View" do Visual Studio, na barra de menu superior, e selecionaremos a opção "SQL Server Object Explorer". Teremos uma estrutura contendo a pasta "Databases", clicaremos sobre ela com o botão direito do mouse e, em seguida, sobre "Add New Database".

Daremos a ele o nome de CasaDoCodigo e, assim, teremos nosso banco de dados vazio. Precisamos passar as informações do banco para a nossa aplicação, por exemplo, qual o servidor e o seu nome. Por enquanto, no banco CasaDoCodigo temos somente as tabelas do sistema, para criarmos nossas próprias, precisaremos da string de conexão deste.

Clicaremos sobre o nome do banco e pressionaremos a tecla "F4" para abrirmos a janela de propriedades. Em Connection string, temos o texto contendo a string de conexão. Copiaremos este texto. No projeto "CasaDoCodigo", temos uma maneira padrão do ASP.NET Core de definir as configurações de onde acessar este banco de dados.

Abriremos o arquivo **appsettings.json**, na pasta do projeto "CasaDoCodigo". Nele, há somente uma configuração de log. Portanto, onde será criada a configuração de banco de dados? Teremos uma nova entrada, que chamaremos de **ConnectionStrings:**

**{**

**"Logging": {**

**"IncludeScopes": false,**

**"LogLevel": {**

**"Default": "Warning"**

**}**

**},**

**"ConnectionStrings": {**

**}**

**}**

**Inseriremos a conexão default, que terá este nome e, por padrão, faremos a conexão no banco de dados que acabamos de criar:**

{

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Warning"

}

},

"ConnectionStrings": {

**"Default"**: "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=CasaDoCodigo;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False"

}

}

**Com isso, o ASP.NET Core terá informações necessárias para poder acessar o banco. Retornaremos à aplicação, para definirmos como utilizar a ConnectionString. No projeto, temos uma classe de configuração chamada Startup.cs. Nela, há dois métodos principais, o ConfigureServices() e o Configure(). Abriremos o ConfigureServices() e definiremos um método de configuração para o nosso banco de dados.**

Temos um serviço já existente, que é o MVC. Para adicionarmos um novo, chamaremos services com o método AddDbContext, passando o nome da classe do contexto do banco de dados, no caso, o ApplicationContext().

Faremos a chamada para este último método passando as opções de configuração do contexto como argumento, ou seja, a definição da utilização da ConnectionString. Assim, passaremos o nome do parâmetro, seguido pelo operador lâmbda **(options => options.UseSqlServer(connectionString)).**

A seguir, chamaremos o parâmetro options com o método que nos permite utilizar, como nosso banco de dados, o SQL Server. Precisamos importar o namespace correspondente, utilizando o atalho "Ctrl + ." sobre UseSqlServer. Por fim, passaremos como parâmetro a ConnectionString.

**namespace CasaDoCodigo**

**{**

**public class Startup**

**{**

**public Startup(IConfiguration configuration)**

**public IConfiguration Configuration { get; }**

**// This method gets called by the runtime. Use this method to add services to the container.**

**public void ConfigureServices(IServiceCollection services)**

**{**

**services.AddMvc();**

//Adicionando o banco de dados, pode ser SQLServer, MySQL, Oracle, etc. (Entity framework)

**services.AddDbContext<ApplicationContext>(options =>**

**options.UseSqlServer(connectionString));**

**}**

**// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.**

**public void Configure(IApplicationBuilder app, IHostingEnvironment env, IServiceProvider serviceProvider){**

**}**

**}**

**Feito isso, a ConnectionString ainda não foi reconhecida. Na realidade, esta variável sequer existe em nosso código. Por isso, precisamos criá-la, buscando as configurações presentes no arquivo AppSettings.json. Acessaremos o objeto Configuration e o método GetConnectionString(), passando o nome da configuração, que no nosso caso é Default:**

**namespace CasaDoCodigo**

**{**

**public class Startup**

**{**

**public Startup(IConfiguration configuration)**

**public IConfiguration Configuration { get; }**

**public void ConfigureServices(IServiceCollection services)**

**{**

**services.AddMvc();**

**string connectionString = Configuration.GetConnectionString("Default");**

**services.AddDbContext<ApplicationContext>(options =>**

**options.UseSqlServer(connectionString)**

**);**

**}**

**public void Configure(IApplicationBuilder app, IHostingEnvironment env)**

**}**

**}**

**Assim, geramos o banco de dados a partir das classes do nosso modelo, com base nesta configuração.**

***Você acabou de desenvolver o modelo de dados e também o contexto do Entity Framework Core através da classe ApplicationContext, que implementa a classe-base DbContext, para realizar as operações para obter/gravar dados no seu banco de dados Sql Server através das entidades da aplicação.***

***O que você precisa fazer para adicionar esse contexto do Entity Framework aos serviços do ASP.NET Core?***

**Chamar o método de extensão**

***services.AddDbContext<ApplicationContext>(options) dentro do método Startup.ConfigureServices.***

***Isso aí! Devemos chamar o método AddDbContext<> dentro do método Startup.ConfigureServices:***

***services.AddDbContext<ApplicationContext>(options =>***

***options.UseSqlServer(<<string de conexão>>)***

***);***

**Resumo 3:** De um lado, temos as classes do modelo em C# e, do outro, o banco de dados vazio. Portanto, precisamos fazer a geração das tabelas a partir do modelo, e para isso utilizamos o Entity Framework Core. Acessaremos os seus comandos, clicando na opção "Tools", na barra de menu superior. Na sequência, selecionaremos "NuGet Package Manager > Package Manager Console".

No console, podemos digitar os comandos que permitirão gerar tabelas. Criaremos um pacote de atualizações do esquema do banco de dados e, em seguida, o aplicaremos. Ele é chamado de migração, ou migration. Ele contém informações de quais tabelas geraram quais chaves, ou colunas, a partir do código. Para isso, utilizaremos o comando Add-Migration e, em seguida, teremos que inserir o nome da migração. Daremos o nome Inicial.

**PM> Add-Migration Inicial**

Pressionaremos a tecla "Enter" para rodarmos a aplicação, e ele gerará o pacote de migração. Ao finalizar, abriremos o projeto, e na pasta "Migrations" observamos que há uma classe contendo a data, seguida do nome Inicial.cs, nela temos todo o esquema que foi gerado a partir da entidade Produto, são instruções para a criação da tabela de produtos. Retornaremos ao console. Uma vez que geramos a migração, precisamos fazer a aplicação dela. Para isso, utilizamos o comando Update-Database, seguido de -verbose, que indica ao Entity que ele deve gerar uma série de logs, para podermos visualizar o que está acontecendo. Pressionaremos a tecla "Enter" para executarmos. Os logs foram gerados, assim como a respectiva tabela no banco de dados, como podemos observar em "SQL Server Object Explorer > CasaDoCodigo > Tables > dbo.Produto". Clicaremos com o botão direito do mouse sobre ela, e selecionaremos a opção "View Designer". Podemos visualizar que foi gerada uma tabela contendo uma chave primária id, que é uma das propriedades da nossa classe e, em seguida, mais três linhas, com código, nome e preço do produto. Assim, concluímos a criação da nossa primeira migração. Retornaremos ao ApplicationContext para fazermos o mapeamento do restante das entidades. Assim como fizemos anteriormente, utilizaremos o modelBuilder:

**namespace CasaDoCodigo {**

**public class ApplicationContext : DbContext**

**{**

**public ApplicationContext(DbContextOptions options) : base(options){}**

**protected override void OnModelCreating(ModelBuilder modelBuilder)**

**{**

**base.OnModelCreating(modelBuilder);**

**modelBuilder.Entity<Produto>().HasKey(t => t.Id);**

**modelBuilder.Entity<Pedido>().HasKey(t => t.Id);**

**modelBuilder.Entity<ItemPedido>().HasKey(t => t.Id);**

**modelBuilder.Entity<Cadastro>().HasKey(t => t.Id);**

**}**

**}**

**}**

Agora que relacionamos, fizemos o mapeamento de todas as entidades, precisamos descrever como serão os relacionamentos. Utilizaremos os comandos para criar estas conexões, como o HasMany(), acompanhado da expressão lambda t.Itens, indicando que queremos trabalhar com muitos itens. Por fim, vamos inserir o novo método WithOne(), que indica o relacionamento de volta, ou seja, cada item de pedido se relaciona a um pedido individual, relação de um para muitos:

**namespace CasaDoCodigo**

**{**

**public class ApplicationContext : DbContext**

**{**

**public ApplicationContext(DbContextOptions options) : base(options)**

**{**

**}**

**protected override void OnModelCreating(ModelBuilder modelBuilder)**

**{**

**base.OnModelCreating(modelBuilder);**

**modelBuilder.Entity<Produto>().HasKey(t => t.Id);**

**modelBuilder.Entity<Pedido>().HasKey(t => t.Id);**

// Relação de um para muitos

**modelBuilder.Entity<Pedido>()**

**.HasKey(t => t.Id).HasMany(t => t.Itens)**

**.WithOne(t => t.Pedido);**

//---------------------------

**modelBuilder.Entity<ItemPedido>().HasKey(t => t.Id);**

**modelBuilder.Entity<Cadastro>().HasKey(t => t.Id);**

**}**

**}**

**}**

**A seguir, faremos o relacionamento entre o pedido e o cadastro. A diferença é que esta relação se dá de um para um, ou seja, cada pedido está relacionado a um cadastro e vice-versa. Acrescentaremos ainda um método indicando que o atributo é obrigatório, que é o IsRequired().**

**namespace CasaDoCodigo**

**{**

**public class ApplicationContext : DbContext**

**{**

**public ApplicationContext(DbContextOptions options) : base(options)**

**{**

**}**

**protected override void OnModelCreating(ModelBuilder modelBuilder)**

**{**

**base.OnModelCreating(modelBuilder);**

**modelBuilder.Entity<Produto>().HasKey(t => t.Id);**

**modelBuilder.Entity<Pedido>().HasKey(t => t.Id);**

**// Relação de um para muitos, pedido - itens**

**modelBuilder.Entity<Pedido>()**

**.HasKey(t => t.Id).HasMany(t => t.Itens)**

**.WithOne(t => t.Pedido);**

**//Relação de um para um, pedido - cadastro**

**modelBuilder.Entity<Pedido>()**

**.HasKey(t => t.Id).HasOne(t => t.Cadastro)**

**.WithOne(t => t.Pedido).IsRequired();**

**modelBuilder.Entity<ItemPedido>().HasKey(t => t.Id);**

**modelBuilder.Entity<Cadastro>().HasKey(t => t.Id);**

**}**

**}**

**}**

Dando continuidade, faremos isso com o ItemPedido. Cada um estará relacionado a um Pedido e a um Produto. Da mesma forma, o Cadastro está relacionado a um Pedido.

**namespace CasaDoCodigo**

**{**

**public class ApplicationContext : DbContext**

**{**

**public ApplicationContext(DbContextOptions options) : base(options)**

**{**

**}**

**protected override void OnModelCreating(ModelBuilder modelBuilder)**

**{**

**base.OnModelCreating(modelBuilder);**

**modelBuilder.Entity<Produto>().HasKey(t => t.Id);**

**modelBuilder.Entity<Pedido>().HasKey(t => t.Id);**

**modelBuilder.Entity<Pedido>()**

**.HasKey(t => t.Id).HasMany(t => t.Itens)**

**.WithOne(t => t.Pedido);**

**modelBuilder.Entity<Pedido>()**

**.HasKey(t => t.Id).HasOne(t => t.Cadastro)**

**.WithOne(t => t.Pedido).IsRequired();**

**modelBuilder.Entity<ItemPedido>().HasKey(t => t.Id);**

**modelBuilder.Entity<ItemPedido>().HasOne(t => t.Pedido);**

**modelBuilder.Entity<ItemPedido>().HasOne(t => t.Produto);**

**modelBuilder.Entity<Cadastro>().HasKey(t => t.Id);**

**modelBuilder.Entity<Cadastro>().HasOne(t => t.Pedido);**

**}**

**}**

**}**

Com isso, concluímos o mapeamento e podemos criar uma nova migração. Retornaremos ao console e digitaremos o comando Add-Migration Modelo. Pressionaremos a tecla "Enter" para executarmos. O novo pacote será criado. Aplicaremos a nova migração com o comando Update-Database -verbose, pressionando a tecla "Enter" para executá-la.

Agora, ao abrir o SQL Object Explorer vemos as tabelas Cadastro, ItemPedido, Pedido e Produto.

**Resumo 4:** Recapitulando, aprendemos a criar um esquema de banco de dados com tabelas, colunas e chaves a partir das classes do C#, utilizando o Entity Framework **Core. Caso queira se aprofundar nesta ferramenta, você pode acessar o curso** [**Entity Framework Core: Banco de dados de forma eficiente**](https://www.alura.com.br/curso-online-entity-framework-core).

Retornaremos ao projeto e deletaremos o banco de dados "CasaDoCodigo". A seguir, o recriaremos com as tabelas que já estão definidas. Faremos isso aplicando o comando Update-Database -verbose no console. Executando, o banco de dados é recriado, assim como as tabelas. Ao abrirmos o SQL Server Object Explorer novamente, atualizado, veremos que o banco de dados aparece normalmente.

Mas e se quiséssemos criar o banco de dados, assim que a aplicação fosse executada, com o comando "F5", sem precisar digitar o comando Update-Database?

Para fazermos isso, acessaremos a classe Startup.cs, ela que define a configuração da nossa aplicação. Ela possui um método Configure(), que é executado quando a aplicação subir para o servidor, em resposta ao comando "F5". Ela então estará sujeita às requisições do browser e, neste momento, teremos a garantia de que o banco de dados foi criado.

Portanto, acessaremos a classe Database a partir do método Configure(), para garantirmos que ela esteja criada. O primeiro passo será modificar a assinatura deste método, para podermos adicionar um novo parâmetro. Após IHostingEnvironment env, adicionaremos uma vírgula (,) e quebraremos uma linha, para incluirmos o IServiceProvider. Como o nome induz, ele fornecerá um serviço para nós, que é o contexto da aplicação, do banco de dados, chamado ApplicationContext. Definiremos o parâmetro serviceProvider.

**namespace CasaDoCodigo**

**{**

**public class Startup**

**{**

**public Startup(IConfiguration configuration)**

**public IConfiguration Configuration { get; }**

**// This method gets called by the runtime. Use this method to add services to the container.**

**public void ConfigureServices(IServiceCollection services)**

**{**

**services.AddMvc();**

**string connectionString = Configuration.GetConnectionString("Default");**

**services.AddDbContext<ApplicationContext>(options =>**

**options.UseSqlServer(connectionString)**

**);**

**}**

**// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.**

**public void Configure(IApplicationBuilder app, IHostingEnvironment env, IServiceProvider serviceProvider)**

**}**

**}**

Copiaremos e chamaremos o serviceProvider abaixo, para que ele nos gere uma nova instância de ApplicationContext. Então acessaremos, a partir do contexto, o Database. Assim, este objeto tem o método necessário para garantir que ele gere o banco de dados, caso este ainda não exista, que é o EnsureCreated().

**namespace CasaDoCodigo**

**{**

**public class Startup**

**{**

**public Startup(IConfiguration configuration)**

**public IConfiguration Configuration { get; }**

**// This method gets called by the runtime. Use this method to add services to the container.**

**public void ConfigureServices(IServiceCollection services)**

**{**

**services.AddMvc();**

**string connectionString = Configuration.GetConnectionString("Default");**

**services.AddDbContext<ApplicationContext>(options =>**

**options.UseSqlServer(connectionString)**

**);**

**}**

**// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.**

**public void Configure(IApplicationBuilder app, IHostingEnvironment env, IServiceProvider serviceProvider)**

**{**

**if (env.IsDevelopment())**

**{**

**app.UseBrowserLink();**

**app.UseDeveloperExceptionPage();**

**}**

**else**

**{**

**app.UseExceptionHandler**

**}**

**app.UseStaticFiles();**

**app.UseMvc(routes =>**

**{**

**routes.MapRoute(**

**name: "default",**

**template: "{controller=Pedido}/**

**{action=Carrossel}/{id?}");**

**});**

**serviceProvider.GetService<ApplicationContext>()**

**.Database.EnsureCreated();**

**}**

**}**

Colocaremos um break point nesta linha que acabamos de editar, para vermos o que acontece ao executarmos a aplicação, com o atalho "F5". Ao executar, veremos que a aplicação foi iniciada, subiu, e passará pelo serviceProvider para garantir que o banco de dados foi criado.

Abriremos o SQL Server Object Explorer, onde temos o banco de dados "CasaDoCodigo" — o apagaremos. Executaremos novamente a aplicação. Retornando ao explorador do SQL e atualizando-o, veremos que o banco de dados foi recriado, com todas as suas tabelas.

Portanto, o que vimos foi uma configuração da nossa aplicação do ASP.NET Core, a partir do método Configure(). Resumindo, temos a classe Startup, que define o modo como configuramos a aplicação. Há nela dois métodos, o ConfigureServices() e o Configure().

O **ConfigureServices()** serve para adicionarmos serviços, por exemplo, o SQL Server, ou o serviço de log. Já a classe **Configure()** é onde o serviço é consumido, ou utilizado. Por esse motivo, este método também é chamado de configuração de **pipeline**. Com ele, podemos indicar, por exemplo, que nossa aplicação utilizará o MVC.

O método Configure() define, por exemplo, que vamos utilizar arquivos estáticos em nossa aplicação, por meio do UseStaticFiles(). Já a utilização do MVC é definida pelo método UseMvc(), ou ainda, podemos configurar a chamada do método que acabamos de criar, para garantir que o banco de dados tenha sido criado. Isto é feito pelo parâmetro novo serviceProvider.

Como o ASP.NET Core MVC reconheceu o parâmetro novo? Ele fez isso graças a uma técnica chamada injeção de dependência. Ele já possui esta técnica nativamente, e pode ser utilizada para criar instâncias a partir da definição de parâmetros do tipo interface. Ou seja, podemos definir uma interface, dizer que ela gerará uma instância de uma determinada classe, e então injetar, inserindo parâmetros para a criação das novas instâncias. Também podemos utilizar um esquema alternativo de injeção de dependência, utilizando um outro framework, como o ***ninject***.

**Injeção de dependência:**

**// Esse serviço garante que o banco de dados foi criado, se não foi criado ele vai mapear todo o código**

**// e criar um novo banco de dados.**

**serviceProvider.GetService<AplicationContext>().Database.Migrate();**

Vamos complementar as aulas anteriores, fazendo uma breve observação. Anteriormente, vimos como criar um banco de dados com o método EnsureCreated(). Contudo, este método não utiliza migrações. Sendo assim, como ele cria o banco de dados? Verificando se o banco ainda não existe, em caso positivo, ele procura o nosso modelo e mapeamento, para poder fazer um esquema, e então gerar o banco de dados do SQL Server.

Contudo, uma vez que utilizamos o método EnsureCreated(), não podemos mais aplicar nenhuma migração no sistema. Portanto, o recomendável é utilizar, em seu lugar, o método **Migrate(),** que faz a mesma coisa, só que utilizando as migrações. O banco é criado da mesma forma, só que utilizando migrações para isso e, ao final, permite que adicionemos novas migrações, caso necessário.

**Diferença entre EnsureCreated() e Migrate():**

1 - O método **EnsureCreated()** cria o banco de dados e seus objetos (tabelas, campos, índices, etc) usando apenas as entidades e mapeamentos da aplicação. Já o método **Migrate()** realiza exatamente a mesma tarefa, porém usando apenas as migrações da aplicação.

- Isso mesmo! Além disso, uma vez que o banco de dados foi gerado através do método **EnsureCreated(),** a aplicação não pode mais usar migrations. Por isso, a não ser que uma aplicação seja pequena e destinada a testes, é recomendável usar sempre o método **Migrate().**

**Em outras palavras o EnsureCreated() não deixa criar outras migrações depois que um Banco de dados é criado por esse método, por outro lado, o método Migrate() garante as mesmas funcionalidades do EnsureCreated(), com exceção das criações de migrations. O Migrate() cria o banco de dados, caso ele não exista, percorrendo todo o código, depois de criado o BD, o desenvolvedor pode utilizar outras migrações.**

**Para Saber Mais...**

**Como Ler Configurações no Controller?**

Você pode ter se deparado com esse problema da vida real, fora do contexto do nosso curso: **como ler configurações do ASP.NET Core 2.0 a partir de um controller?** Vamos partir do pressuposto de que você tenha uma aplicação de blog com uma estrutura de configuração definida no arquivo appsetings.json.

Vamos começar pelo arquivo appsetings.json. Digamos que sua configuração seja definida pelo arquivo abaixo:

**{**

**"ConnectionStrings": {**

**"Blog": "Server=(localdb)\\mssqllocaldb;Database=MeuBlog;Trusted\_Connection=True;"**

**},**

**"Security": {**

**"Language" : "pt-BR",**

**"SuperUser": {**

**"Login": "bgiorgione",**

**"Email": "bgiorgione@bgiorgione.com.br",**

**"ShowEmail": "true"**

**},**

**"Admin": {**

**"Login": "moliveira",**

**"Email": "mclricardo@gmail.com.br",**

**"ShowEmail": "false"**

**}**

**}**

**}**

*Listagem: o arquivo appsetings.json*

Todos os método de que precisamos precisam vir da classe Configuration, que é uma implementação da interface IConfiguration.

O primeiro passo **seria** configurar o *mecanismo de injeção de dependência* do ASP.NET Core para obter o objeto de configuração. Mas felizmente esse objeto já vem automaticamente registrado internamente no contêiner de injeção de dependência do ASP.NET Core, portanto você **não precisa** adicionar o código services.AddSingleton<IConfiguration>(Configuration); na sua classe Startup.

Agora, como vimos no curso, basta adicionar um parâmetro IConfiguration no construtor do controller para começarmos a obter as configurações:

**using Microsoft.AspNetCore.Mvc;**

**using Microsoft.Extensions.Configuration;**

**namespace Blog.Controllers**

**{**

**public class BlogController : Controller**

**{**

**private IConfiguration \_configuration;**

**public BlogController(IConfiguration Configuration)**

**{**

**\_configuration = Configuration;**

**}**

**public IActionResult Usuarios()**

**{**

**// Configuração usando a string de chave**

**ViewData["SecurityLanguage"] = \_configuration["Security:Language"];**

**// Configuração usando método GetSection**

**ViewData["SecuritySuperUserLogin"] =**

**\_configuration.GetSection("Security").GetSection("SuperUser").GetSection("Login");**

**// Configuração usando GetSection e string de chave ao mesmo tempo**

**ViewData["SecuritySuperUserEmail"] =**

**\_configuration.GetSection("Security")["SuperUser:Email"];**

**// Configuração usando GetSection e string de chave ao mesmo tempo**

**ViewData["SecuritySuperUserShowEmail"] =**

**\_configuration.GetSection("Security")["SuperUser:ShowEmail"];**

**return View();**

**}**

**}**

**}**

*Listagem: o arquivo BlogController.cs*

No exemplo acima, populamos o ViewData com as configuração. Mas você pode utilizar as configurações como quiser, claro.

Agora basta exibir os dados na view Usuarios:

**<p>**

**SecurityLanguage <strong>@ViewData["SecurityLanguage"]</strong><br />**

**SecuritySuperUserLogin**

**<strong>@ViewData["SecuritySuperUserLogin"]</strong><br />**

**SecuritySuperUserEmail**

**<strong>@ViewData["SecuritySuperUserEmail"]</strong>**

**SecuritySuperUserShowEmail**

**<strong>@ViewData["SecuritySuperUserShowEmail"]</strong>**

**</p>**

**Listagem: o arquivo Usuarios.cshtml**

# Para saber mais...

### Diferenças entre o Entity Framework 6 e o Entity Framework Core

Neste curso utilizamos **Entity Framework Core 2.0** como ORM (Framework de Mapeamento Objeto-Relacional). Talvez você esteja se perguntando: "Quais as diferenças entre o Entity Framework Core e o Entity Framework 6 que eu costumava usar em projetos anteriores?"

### Primeiro caso: Para aplicações EF6 existentes

O Entity Framework Core é considerado um avanço em relação ao Entity Framework 6, porém **nem todas as funcionalidades** do EF6 foram implementadas no EF Core. As próximas versões do EF Core irão tratar de eliminar essas limitações.

É preciso um bom motivo para migrar um projeto EF 6 para EF Core, caso contrário você pode estar arriscando seu projeto.

Veja no final da página as tabelas comparativas entre as duas versões do Entity Framework para decidir qual a melhor opção.

### Segundo caso: Para novas aplicações

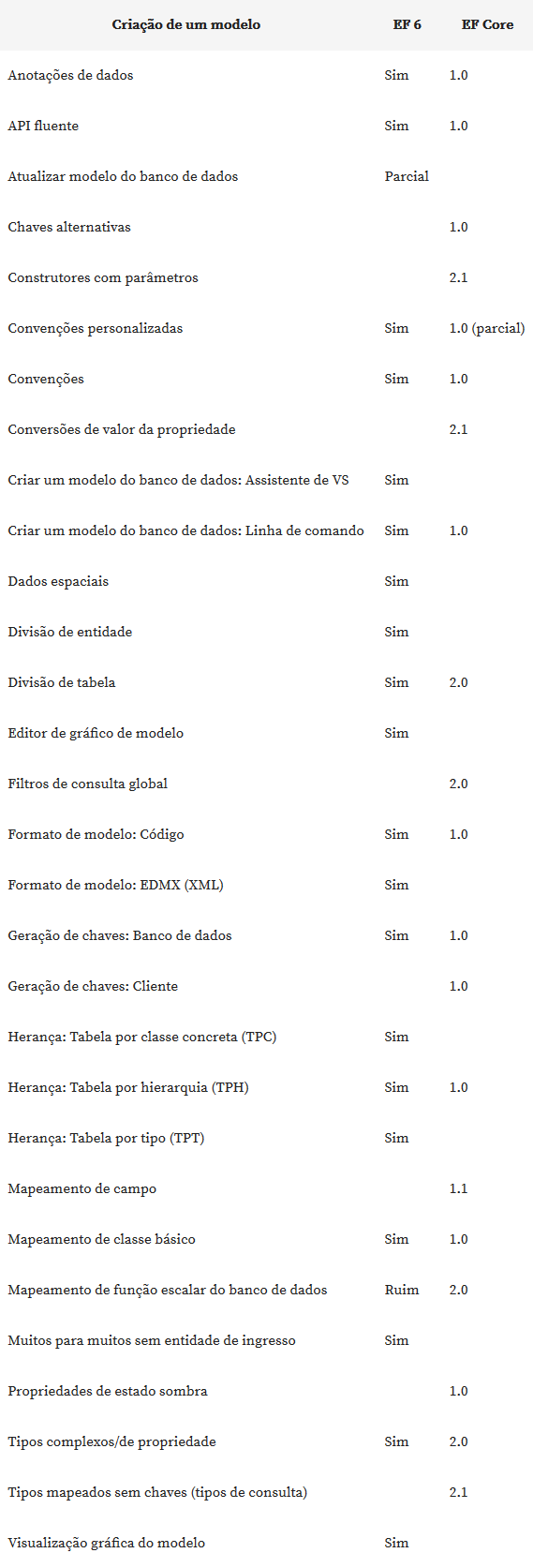
Se você precisar criar novas aplicações, é recomendával utilizar o EF Core, desde que seu aplicativo não necessite de recursos do Entity Framework 6 que ainda não foram implementados no EF Core.

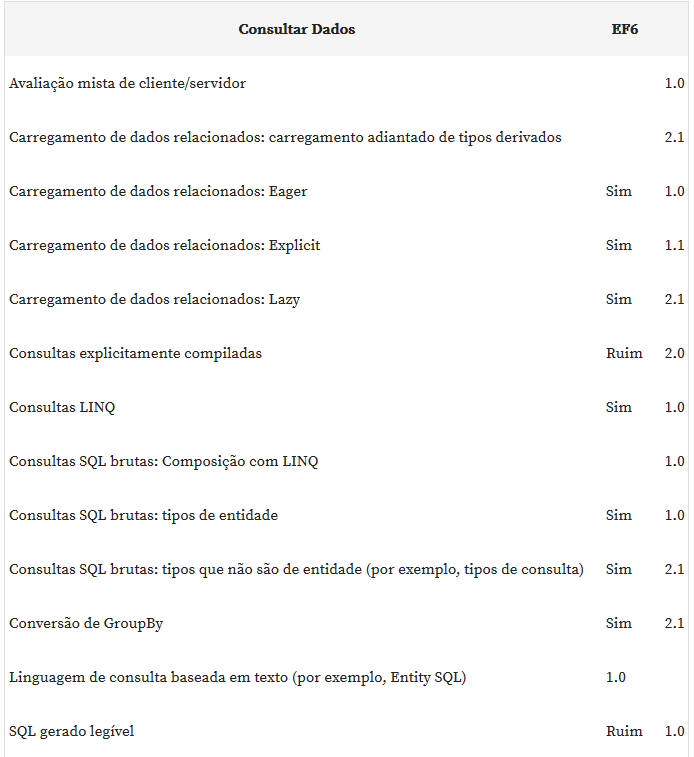
O EF 6 **depende do Windows**. Isso significa que você não poderá criar novas aplicações com EF 6 para o .NET Core. Somente o Entity Framework Core pode rodar sobre o .NET Core.

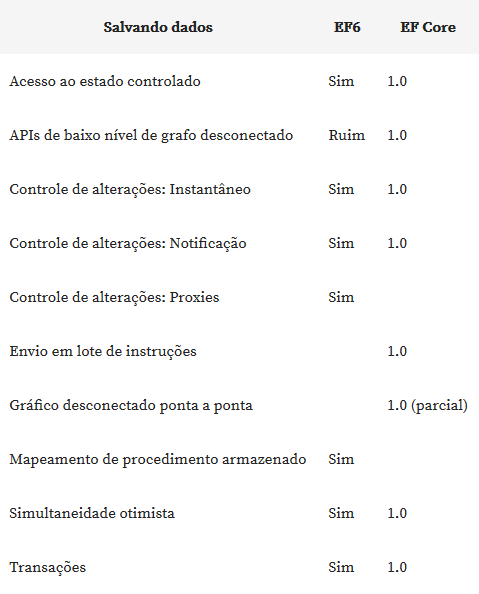
Você ainda pode utilizar o Entity Framework 6 em novos projetos, porém é importante saber se ele é aceitável para sua aplicação. Você pode visualizar as diferenças entre EF 6 e EF Core nas tabelas abaixo, que podem lhe ajudar a decidir qual das duas opções é mais adequada ao seu projeto.

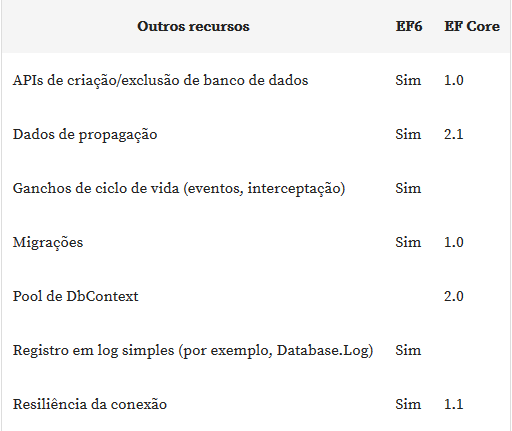
## Comparação entre os recursos do EF6 e EF Core

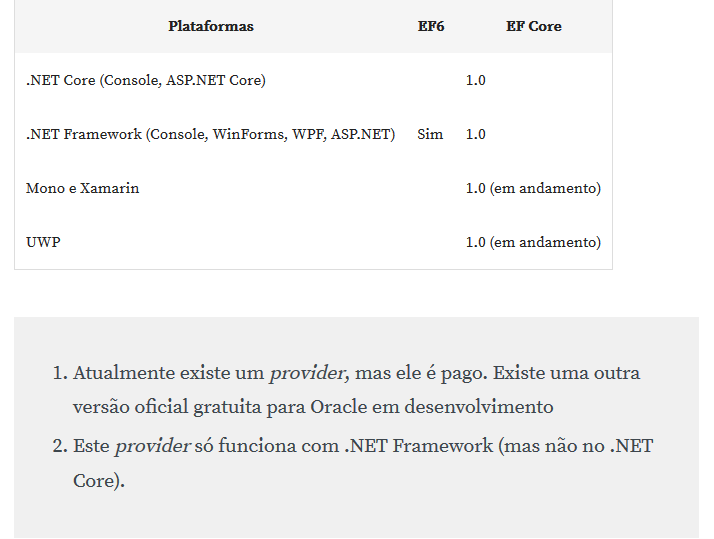
As informações se referem à primeira versão em que o recurso aparece. Quando o recurso não é suportado, a coluna está em branco.









Nesse capítulo, aprendemos:

* Criando o Modelo de dados
* Configurando o Banco de Dados SQL Server para uso
* Relacionamentos e Migrações
* Gerando Banco, Classe Startup, Serviços, Injeção de Dependências
* Método EnsureCreated X Método Migrate

No link a seguir, você pode fazer o [DOWNLOAD completo do projeto deste capítulo](https://github.com/alura-cursos/ASPNETCore20/archive/e0982e5e737cacd7dc0058cabd03e9989914c914.zip) para acompanhar com o vídeo ou comparar seu código com as vídeo-aulas.

**Aula 4 – Resumo 1:** Poderíamos aproveitar que estamos criando o banco de dados automaticamente para implementarmos um método para a carga destes dados, e alimentarmos nosso banco com o catálogo inicial de produtos. Porém, com isso estaríamos violando o princípio de **separação de responsabilidades**.

Sendo assim, criaremos uma classe específica (DataService) para não poluirmos a classe Startup, com um método público denominado InicializaDB() para inicializar o banco de dados, caso seja necessário. Para isso, ele precisará acessar o contexto, ApplicationContext.

Portanto, criaremos um campo privado e, para que o contexto seja inicializado, criaremos também uma instância. Poderíamos usar simplesmente private readonly ApplicationContext contexto = new ApplicationContext, mas não o faremos porque queremos usar o sistema de injeção de dependências do ASP.NET Core.

Selecionaremos o campo privado contexto usando o atalho "Ctrl + ." e gerando um construtor, que tomará o parâmetro ApplicationContext contexto como obrigatório na criação do objeto DataService, e então teremos a instância do contexto:

**class DataService**

**{**

**private readonly ApplicationContext contexto;**

**public DataService(ApplicationContext contexto)**

**{**

**this.contexto = contexto;**

**}**

**public void InicializaDB()**

**{**

**contexto.Database.EnsureCreated();**

**}**

**}**

Por padrão, o ApplicationContext é gerado pela injeção de dependência, assim, é possível acessar o objeto contexto no método InicializaDB(). Para garantir que ele seja criado, usaremos EnsureCreated(), e com isso chamaremos o DataService, em vez de fazer a criação do banco de dados diretamente na classe Startup.

Queremos que se crie uma instância do serviço de dados, para o qual basta substituirmos ApplicationContext por DataService em serviceProvider.GetService<ApplicationContext>().Database.EnsureCreated(), mais acima no código.

Em seguida, chamaremos o novo método, InicializaDB(), deixando a linha de código da seguinte maneira: serviceProvider.GetService<DataService>().InicializaDB(). Porém, para o serviço de injeção de dependência do ASP.NET Core funcionar, é necessário registrar a nova classe, isto é, DataService, no contêiner de injeção de dependências.

Iremos ao método ConfigureServices() da classe Startup e faremos o registro chamando um método que adiciona uma instância, a qual queremos que exista somente enquanto os objetos que a utilizarem estiverem ativos. Este método é AddTransient, que significa adicionar uma instância temporária.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

string connectionString = Configuration.GetConnectionString("Default");

services.AddDbContext<ApplicationContext>(options =>

options.UseSqlServer(connectionString));

services.AddTransient<DataService>();

}

Normalmente, trabalhamos com interfaces para a injeção de independência, então moveremos a classe DataService para um novo arquivo utilizando "Ctrl + ." e escolhendo a opção "Move type to DataService.cs". Vamos acessá-lo pelo painel de "Solution Explorer", do lado direito.

Criaremos uma interface a partir deste novo arquivo, com "Ctrl + ." e "Extract Interface...". Na janela que se abre, clicaremos em "OK", e a interface será extraída. Vamos voltar a Startup.cs e, no registro da injeção de dependência, incluiremos o tipo da interface: services.AddTransient<IDataService, DataService>();.

# Resumo 2:

# O que é injeção de dependência e por que ela é necessária?

Vamos dar uma olhada no código do PedidoController:

public class PedidoController : Controller

{

private readonly IProdutoRepository produtoRepository;

public PedidoController(IProdutoRepository produtoRepository)

{

this.produtoRepository = produtoRepository;

}

...

Note que esse construtor está exigindo um parâmetro: public PedidoController(IProdutoRepository produtoRepository). Esse parâmetro é uma **dependência**, isto é, sem ele o controller não vai funcionar, pois precisamos do repositório de produtos para podermos consultar os produtos do e-commerce.

Mas como esse parâmetro é injetado? Note que, **em nenhum momento**, nós instanciamos um controller diretamente, como por exemplo PedidoController meuController = new PedidoController(), certo? Quando chamamos uma url para acessar action através do navegador, o controller é instanciado pelo próprio ASP.NET Core, por baixo dos panos. O que fazemos na injeção de dependência é informar o ASP.NET Core **sobre tudo o que a instância do controller precisa para ser criada**. E então o que o ASP.NET Core faz automatimente, sem a gente ver?

* Ele cria uma instância de IProdutoRepository (através da classe ProdutoRepository)
* Ele cria uma instância de PedidoController, passando a instância de IProdutoRepository criada no passo 1.

Ou seja, esse parâmetro é injetado automaticamente sempre que o ASP.NET Core cria uma nova instância do PedidoController.

Qual seria a alternativa a isso?

Sem injeção de dependência, teríamos que criar as instâncias manualmente, usando o **operador new**. Teríamos que informar a **classe concreta**, em tempo de desenvolvimento, o tempo todo. Com injeção de dependência, podemos simplesmente definir parâmetros nos construtores, que são **interfaces e não classes concretas**. Por que isso é importante? Porque quando programamos com interfaces fazemos com que as classes **não precisem conhecer** as classes concretas que vão ser criadas, apenas as interfaces. Isso é bom porque **diminui o acoplamento** e a dependência entre classes.

Um exemplo de injeção de dependência: Imagine que você tenha uma aplicação que gera logs sobre a atividade do usuário no seu site. Você então utiliza injeção de dependência, fazendo seu programa chamar métodos da interface IUserLog. Inicialmente, você define que os logs vão ser gravados em arquivo texto, através de uma classe FileLog (que implementa IUserLog). Mas depois de um tempo, você decide gravar os logs em banco de dados, usando uma outra classe chamada SQLServerLog (que também implementa IUserLog).

Como você faria essa mudança? Com injeção de dependência, você pode simplesmente mudar uma única linha de configuração, indicando que IUserLog deve ser instanciada como SQLServerLog, e não mais como FileLog. Agora, **sem injeção de dependência** você teria que substituir todas as chamadas da classe FileLog para SQLServerLog.

# Resumo 3:

Ao rodarmos nossa aplicação novamente, percebemos que cometemos um engano ao obtermos uma instância do DataService — o ASP.NET Core não reconhece este tipo. Isso porque na verdade criamos o registro de uma interface IDataService, sendo necessário trocarmos o GetService:

serviceProvider.GetService<IDataService>().InicializaDB();

Isso, sim, fará com que seja gerada uma instância a partir desta interface. Acessaremos InicializaDB() da classe DataService e começaremos a fazer a carga do nosso catálogo de produtos. Felizmente, já temos um arquivo JSON contendo as informações acerca destes produtos.

Na pasta "\_Recursos" da Solução há uma subpasta chamada "dados", que por sua vez contém "livros.json", com código, nome e preço de cada um dos produtos, no caso, livros. Faremos uma cópia deste arquivo no nosso projeto "CasaDoCodigo".

Como iremos ler este arquivo .json?

Poderemos utilizar a classe File do .NET para acessá-lo. Importaremos a referência (o *namespace*), e faremos a leitura de todo o texto do arquivo, com ReadAllText(). Passaremos o nome do arquivo como parâmetro deste método e precisaremos armazenar o texto em uma variável denominada json.

Para realizarmos a conversão deste JSON em uma lista de objetos a serem utilizados para alimentar o banco de dados, usaremos uma biblioteca bem conhecida no .NET, o ***Newtonsoft***. Para isto, chamaremos a classe desta biblioteca e o método para desserializar e transformar o texto em objetos, passando o tipo para o qual queremos converter como parâmetro, mas antes disso criaremos um tipo chamado Livro.

Copiaremos os campos do arquivo livros.json e criaremos três propriedades automáticas para esta nova classe, definindo-os com { get; set; }. Feito isto, utilizaremos Livro para desserializar o arquivo JSON, passando o tipo, no caso uma lista de Livro. Passaremos a nossa *string* como parâmetro, que é a variável json, e então armazenaremos este conteúdo em uma lista, um novo objeto, chamado livros. Incluiremos também um *breakpoint* na última linha do método.

class DataService : IDataService

{

private readonly ApplicationContext contexto;

public DataService(ApplicationContext contexto)

{

this.contexto = contexto;

}

public void InicializaDB()

{

contexto.Database.EnsureCreated();

var json = File.ReadAllText("livros.json");

var livros = JsonConvert.DeserializeObject<List<Livro>>(json);

}

}

class Livro

{

public string Codigo { get; set; }

public string Nome { get; set; }

public decimal Preco { get; set; }

}

Vamos rodar a aplicação para verificar se a conversão é bem sucedida, iniciando pela criação do banco e, em seguida, pela carga do arquivo livros.json. Após a passagem pelo *breakpoint*, na linha 22, acessaremos o conteúdo de livros e veremos as informações de código, nome e preço, como gostaríamos.

Adiante, começaremos a importar estes dados para o nosso banco de dados.

Exercícios: 1

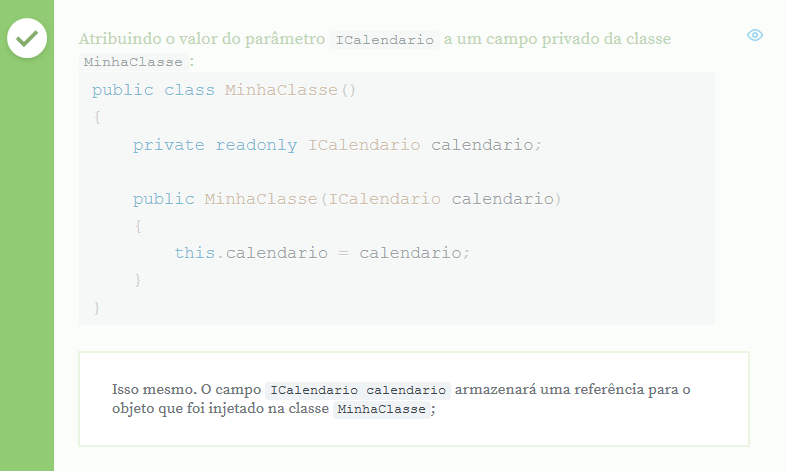
Você está desenvolvendo uma aplicação da área de Recursos Humanos usando ASP.NET Core 2.0 MVC, e acaba de criar uma nova classe chamada Calendario. Essa classe possui métodos que serão úteis para obter dados para várias páginas. Os métodos são:

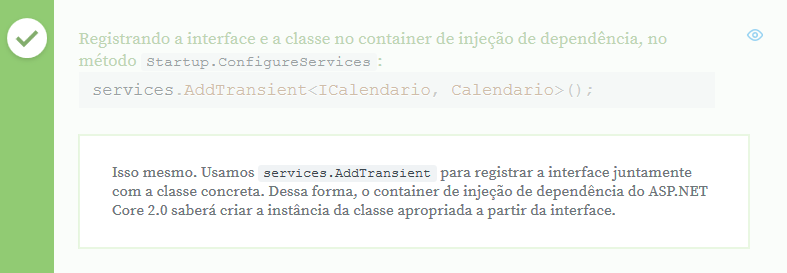
## public int GetDiasCorridos(DateTime dataInicial, DateTime dataFinal)

## public int GetDiasUteis(DateTime dataInicial, DateTime dataFinal)

## public DateTime GetProximasFerias(int funcionarioId)

Você deseja que uma instância da classe Calendario esteja disponível para uso em todos os pontos da sua aplicação, e deseja que o código não referencie a classe diretamente, mas sim através da sua interface ICalendario. Quais os passos necessários para realizar essa tarefa?





Quando definimos que os parâmetros de métodos e construtores de uma determinada classe são **classes concretas**, corremos o risco de *"engessar"* as relações entre as classes.

É uma boa prática **programar para interfaces**, pois isso diminui o *acoplamento* entre as classes, isto é, *diminui* a dependência entre elas. Por exemplo, imagine esta classe:

**public class Automovel()**

**{**

**public Automovel(MotorAGasolina motor)**

**{**

**...**

**}**

**}**

O que acontece quando você precisa trocar MotorAGasolina por uma outra classe, chamada MotorAAlcool, que implementa os mesmos métodos e interfaces? Nesse caso, você precisa mudar a classe Automovel:

**public class Automovel()**

**{**

**public Automovel(MotorAAlcool motor)**

**{**

**...**

**}**

**}**

Mas isso não é bom, pois você está **modificando uma classe que funciona perfeitamente**, apenas para trocar o tipo de um parâmetro!

Isso representa uma violação de um princípio conhecido como **"open-closed principle"**, isto é, **aberto para extensibilidade, fechado para modificação**. De acordo com esse princípio, você deveria projetar sua classe para que as mudanças, como alterações nos tipos de parâmeros, não necessitem de alterações na classe Automovel.

Mas quando você estabelece que a interface utilizada é de uma determinada interface, você já definiu um **contrato** entre a classe Automovel e o tipo de classe que ela recebe como parâmetro do construtor:

**public class Automovel()**

**{**

**public Automovel(IMotor motor)**

**{**

**}**

**}**

A partir daí, não importa qual é exatamente a classe recebida como parâmetro do construtor de Automovel, desde que ela implemente a interface IMotor.

**Resumo 4:** Tendo os objetos — os livros —, começaremos a importá-los como produtos do nosso banco de dados, varrendo-os e inserindo-os um a um com o laço foreach. Assim, para cada livro contido na coleção livros, acessaremos o que seria a tabela do banco de dados, que no *Entity Framework* chamamos de ***DbSet***, e está dentro de contexto.

Passaremos o nome da classe que representa uma tabela do banco, no caso, a tabela Produto, e então chamaremos o método para adicionar uma nova instância de Produto da nossa lista de produtos, passando o código, nome e preço do livro como argumentos.

Entretanto isto ainda não é o suficiente, pois quando adicionamos tudo isso ao DbSet de Produto, o que fazemos é adicionar informações em memória, as quais ainda não são de fato enviadas ao banco de dados, exigindo que isto seja feito todas as vezes em que um produto for adicionado ao DbSet.

Para que isto ocorra de uma vez só, chamaremos o método SaveChanges() a partir de contexto e, aí sim, quando tivermos todos os produtos carregados em memória, prontos para serem adicionados ao banco de dados, fazemos a gravação no banco de dados de uma única vez. Nesta linha, também acrescentaremos um *breakpoint*, e então rodaremos a aplicação novamente.

**public void InicializaDB()**

**{**

**contexto.Database.EnsureCreated();**

**var json = File.ReadAllText("livros.json");**

**var livros = JsonConvert.DeserializeObject<List<Livro>>(json);**

**foreach (var livro in livros)**

**{**

**contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));**

**}**

**contexto.SaveChanges();**

**}**

Recapitulando:

* o serviço será subido;
* quando o arquivo de produtos for lido, cada livro será inserido ao DbSet;
* eles serão enfim gravados no banco de dados, com SaveChanges().

Atualizaremos o SQL Server Object Explorer, e em localdb, temos "Databases > CasadoCodigo > Tables". Clicaremos com o lado direito do mouse em "dbo.Produto", selecionaremos "View Data" e visualizarmos os dados. Então, teremos a carga dos produtos a partir do nosso arquivo livros.json.

Exercício 1: **Vanessa criou um método InicializaDB para ler um arquivo json contendo informações sobre livros e, em seguida, inserir dados iniciais na tabela Produtos através do Entity Framework Core:**

**public void InicializaDB()**

**{**

**contexto.Database.EnsureCreated();**

**var json = File.ReadAllText("livros.json");**

**var livros = JsonConvert.DeserializeObject<List<Livro>>(json);**

**foreach (var livro in livros)**

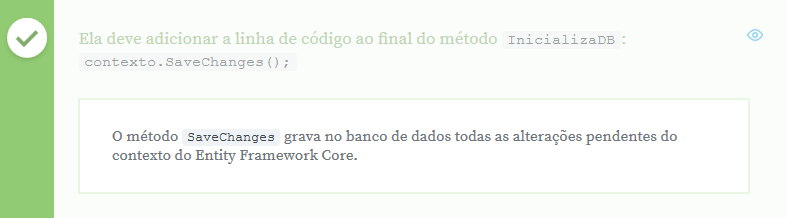
**{**

**contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));**

**}**

**}**

**Quando a aplicação inicia, esse código é executado sem erros, mas em seguida Vanessa consulta o banco de dados e constata que os dados não estão na tabela Produtos. O que Vanessa deve fazer para resolver esse problema?**



**Resumo 5:** A partir de agora, faremos pequenas alterações no nosso código para melhorarmos a qualidade e estrutura do nosso projeto. Temos, por exemplo, o método InicializaDB() que cria e salva produtos no banco de dados, além de ler o arquivo JSON. São muitas responsabilidades para um método só.

Sendo assim, extrairemos para um novo método as seguintes linhas, em que lemos o arquivo JSON. Selecionaremos o trecho e usaremos "Ctrl + ." e "Extract Method". O método será chamado de GetLivros(), e retornará uma lista de livros.

var json = File.ReadAllText("livros.json");

var livros = JsonConvert.DeserializeObject<List<Livro>>(json);

Em seguida, salvaremos os livros no banco de dados, em forma de produtos. E extrairemos o seguinte trecho para o método SaveProdutos():

foreach (var livro in livros)

{

contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));

}

contexto.SaveChanges();

Isto não é o suficiente para melhorarmos o nosso projeto. Na verdade, queremos ter uma classe especializada em ler, gravar, fazer alterações e manipular os dados da entidade de produtos. Para isso, utilizaremos um padrão de projeto de acesso a dados a partir da criação de um diretório. Clicaremos com o botão direito do mouse em "CasaDoCodigo > Add > New Folder" e renomearemos a pasta com "Repositories".

Porém, ao tentarmos nomear, seremos informados de que a pasta já existe. Portanto, clicaremos nela com o lado direito do mouse e em "Include In Project". Com isto, começaremos a criar nosso repositório de produtos, ao qual adicionaremos uma nova classe, ProdutoRepository.

Moveremos nosso método SaveProdutos() para esta nova classe, recortando-o de DataService.cs. Com isso, deixamos de ter acesso ao objeto contexto, porque ele ficou na classe DataService. Então, copiaremos o campo previamente criado para contexto, isto é, private readonly ApplicationContext contexto, e o colaremos na nova classe de repositório.

Em ProdutoRepository.cs teremos, portanto:

namespace CasaDoCodigo.Repositories

{

public class ProdutoRepository

{

private readonly ApplicationContext contexto;

private void SaveProdutos(List<Livro> livros)

{

foreach (var livro in livros)

{

contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));

}

contexto.SaveChanges();

}

}

}

No entanto, a classe Repositories não instanciará o contexto, pois ela o receberá em seu construtor através da injeção de dependência. Criaremos um novo construtor para a nova classe selecionando o campo contexto, e usando "Ctrl + ." e "Generate constructor". Com isso, o contexto será fornecido via injeção de dependência:

private readonly ApplicationContext contexto;

public ProdutoRepository(ApplicationContext contexto)

{

this.contexto = contexto;

}

Vamos mudar a visibilidade de SaveProdutos(), de private para public:

public void SaveProdutos(List<Livro> livros)

{

// foreach omitido

}

O programa indicará um problema em SaveProdutos(), pois Livro se encontra menos acessível do que SaveProdutos(), em DataService.cs. Recortaremos a classe e a colaremos junto à ProdutoRepository, ao fim de seu código e, além disto, usaremos public.

Em seguida, criaremos e extrairemos uma interface a partir da classe ProdutoRepository, usando "Ctrl + ." mais uma vez. Agora sim, a interface no novo arquivo está sendo gerada, IProdutoRepository, com o método SaveProdutos().

Então, voltaremos a DataService.cs para podermos consumir esse método do repositório e, como Livro em List<Livro> livros = GetLivros() foi movido, usaremos "Ctrl + ." para resolvermos a referência. Assim, SaveProdutos() sumirá desta classe, sendo necessário acessarmos o ProdutoRepository, que está em IProdutoRepository.

Para isso, precisaremos registrar a interface IProdutoRepository na classe Startup para podermos utilizar a injeção de dependência. Usaremos services.AddTransient para criarmos uma instância temporária para este tipo, e incluiremos o nome de IProdutoRepository:

public void ConfigureServices(IServiceCollection services)

{

// código omitido

services.AddTransient<IDataService, DataService>();

services.AddTransient<IProdutoRepository, ProdutoRepository>();

}

De volta a DataService.cs e, no local em que temos o construtor, estamos recebendo ApplicationContext contexto. No entanto, precisamos receber o ProdutoRepository no construtor do DataService.cs, então fica faltando criarmos um campo para este repositório, a ser chamado de produtoRepository:

class DataService : IDataService

{

private readonly ApplicationContext contexto;

private readonly IProdutoRepository produtoRepository;

public DataService(ApplicationContext contexto,

IProdutoRepository produtoRepository)

{

this.contexto = contexto;

this.produtoRepository = produtoRepository;

}

// código omitido

}

Utilizaremos este campo privado no DataService para chamar SaveProdutos():

public void InicializaDB()

{

contexto.Database.EnsureCreated();

List<Livro> livros = GetLivros();

produtoRepository.SaveProdutos(livros);

}

Rodaremos a app e constataremos que tudo funciona corretamente. Reabriremos o SQL Server para verificar o carregamento da lista de produtos no banco de dados, selecionando a tabela de produtos (dbo.Produto) em "Databases > CasaDoCodigo > Tables" com o botão direito do mouse e, em "View Data", para visualizarmos os dados.

Os produtos são carregados como esperado, a partir do nosso arquivo JSON, porém desta vez utilizando o repositório de produtos.

**Resumo 6:** Chegou a hora de injetarmos o nosso modelo na *view* para gerarmos uma página dinâmica. E quem faz esta injeção é o *controller*, cujo pedido acessará o banco de dados para passar o modelo à *view* por meio do repositório de produtos.

Assim, modificaremos nosso repositório para podermos criar um método para a obtenção de produtos. Na interface IProdutoRepository, criaremos o método GetProdutos(), a partir do qual retornaremos uma lista contendo todos os produtos do catálogo.

public interface IProdutoRepository

{

void SaveProdutos(List<Livro> livros);

IList<Produto> GetProdutos();

}

Feito isso, iremos à classe ProdutoRepository e forçaremos a implementação desta interface com o novo método. Com IProdutoRepository selecionado, usaremos "Ctrl + ." e optaremos por "Implement interface", e será criado o método GetProdutos(), que acessará o Dbset dos produtos, retornando-o em uma lista:

public IList<Produto> GetProdutos()

{

return contexto.Set<Produto>().ToList();

}

Iremos ao *controller* para consumirmos este método do repositório e pegarmos os produtos. Para isso, acessaremos PedidoController.cs, e criaremos um campo privado que receberá o valor quando fizermos a injeção de dependência. Também criaremos um construtor a partir deste campo, selecionando private readonly IProdutoRepository produtoRepository e usando "Ctrl + .".

public class PedidoController : Controller

{

private readonly IProdutoRepository produtoRepository;

public PedidoController(IProdutoRepository produtoRepository)

{

this.produtoRepository = produtoRepository;

}

}

Agora consumiremos um método de produtoRepository na *action* de Carrossel(), cujo retorno será injetado como parâmetro do método View():

public IActionResult Carrossel()

{

return View(produtoRepository.GetProdutos());

}

Feito isso, modificaremos a *view* de Carrolsel() para que esta listagem seja exibida na tela. Abriremos "Views > Pedido > Carrossel.cshtml" no painel "Solution Explorer", que se encontra do lado direito. Logo após a linha com <h3>Catálogo</h3>, deixaremos um espaço para renderizar a listagem de *views* na tela.

Usaremos **@** para indicar que estamos colocando um código C# em um arquivo .cshtml. Em seguida, incluiremos uma instrução C#, que no caso será a varredura da nossa lista de produtos, o nosso modelo. Em foreach, poderemos colocar um HTML, dentro do qual incluiremos um trecho em C#.

@foreach (var produto in Model)

{

<div>@produto.</div>

}

Porém, o IntelliSense não reconhece ao digitarmos @produto., por não termos descrito o tipo ou modelo desta *view* de carrossel. Teremos que incluir a diretiva @model, que exige o tipo de modelo que iremos injetar na *view*, isto é, a lista de produtos.

@model List<Produto>;

<h3>Catálogo</h3>

@foreach (var produto in Model)

{

<div>@produto.Nome</div>

}

Desta vez, o programa nos mostra as opções ao digitarmos, e então rodaremos a aplicação com a tecla "F5" para verificar a diferença na nossa tela. A parte dos produtos será carregada e exibida na *view*. Com isso, temos o catálogo, e cada um dos produtos contidos no banco de dados.

Adiante, passaremos os produtos para baixo, onde se localiza o carrossel com as informações tais como nome e preço.

**Resumo 7:** Numa view Razor, qual a diferença entre @model (com inicial em minúscula) e @Model (com inicial em maiúscula)?

### Opinião do instrutor:

As duas formas tem uma utilização bem específica:

1) Para declarar com qual **qual modelo** a página Razor irá trabalhar, utilizamos a **diretiva** @model (inicial em minúscula):

@model Cadastro

Nesse caso, Cadastro é a classe que serve de **modelo** para a view.

2) Para **acessar** as propriedades do modelo, precisamos utilizar o objeto Model (inicial em maiúscula):

<input type="text" class="form-control" id="nomeCliente" placeholder="Nome do Cliente"

asp-for="@Model.Nome">

Mas por que nesse caso também foi usado o símbolo de arroba (@) ? Porque ele sempre demarca o **início de código C#** dentro do HTML numa view Razor.

Note que nem sempre o objeto Model é precedido pelo arroba (@): Ele só é necessário quando você está inserindo o Model diretament num código HTML. Se você utilizar o Model dentro de um bloco C#, não se usa o arroba (@):

<div class="carousel-inner" role="listbox">

@{

const int TAMANHO\_PAGINA = 4;

int paginas = (int)Math.Ceiling((double)Model.Count() / TAMANHO\_PAGINA);

}

**Resumo 8:** Vamos colocar os dados acerca do catálogo de produtos no nosso carrossel, dando uma olhada na nossa *view*, com uma estrutura que nos mostra que este HTML fixo foi pensado para que quatro produtos seguidos estejam em um mesmo grupo, denominado "página".

No carrossel, conforme clicamos no ícone branco de seta nas laterais, temos a alternância entre os grupos de quatro produtos sendo exibidos simultaneamente. Representaremos isto modificando o HTML no código para conseguirmos injetar nosso modelo.

Removeremos a div de classe item, ficando apenas com a de item active. E no lugar de quatro produtos, manteremos apenas o primeiro, a ser repetido várias vezes com informações diferentes, isto é, cada vez que ele for exibido, estará associado a um produto diferente.

Portanto, faremos uma estrutura de repetição, o foreach do C#, usando **@**. A repetição terá cada produto em uma página de produtos (produtosDaPagina) mas, antes, incluiremos um código C# com a declaração dessa coleção entre chaves. Pegaremos o modelo e obteremos somente os quatro primeiros elementos desta coleção de produtos, por meio do método Take().

Feito isso, incluiremos a div que representará cada um dos produtos do carrossel dentro do foreach:

<div class="carousel-inner" role="listbox">

<div class="item active">

<div class="row">

@{

var produtosDaPagina = Model.Take(4);

}

@foreach (var produto in produtosDaPagina)

{

// código referente à div omitido

}

E então trocaremos as informações por aquelas do nosso modelo de produtos. Em relação à imagem de cada produto do carrossel, por exemplo, temos um arquivo large\_001.jpg, em que 001 pode ser representado pela propriedade código do nosso produto, sendo substituído portanto por `@produto.Codigo, e ficandolarge\_@produto.Codigo.jpg`.

Da mesma forma, mais abaixo, onde se encontra o nome do produto, ASP.NET Core MVC, trocaremos para `@produto.Nome, e no lugar do preço,R$ 69,90, deixaremosR$ @produto.Preco`. Com isso, poderemos rodar a aplicação, atualizando a página.

Ocorrerá um erro de compilação, pois o programa está considerando que jpg (a extensão do arquivo de imagem), faz parte do código C#. Envolveremos esta parte com parênteses, pois assim o delimitaremos. O trecho ficará assim: large\_@(produto.Codigo).jpg.

Ao rodarmos a aplicação novamente, veremos que o carrossel traz as informações provenientes do nosso modelo para a apresentação de uma página de produtos.

**Resumo 9:** Agora que conseguimos exibir a primeira página do nosso catálogo, tentaremos exibir as demais. Voltaremos à *view* do carrossel, e remover o código do primeiro laço, feito com base nos produtos do modelo. Identificaremos a div de classe item active, que mostra cada uma das páginas do carrossel.

Faremos um laço com código C#, iniciando com **@** e, entre chaves ({}), colocaremos a declaração de uma constante que será o tamanho de cada página, isto é, 4. Em seguida, calcularemos a quantidade de páginas que queremos exibir, para o qual dividiremos a quantidade de produtos pelo tamanho da página.

Porém, esta conta não é o suficiente, pois caso o resultado seja um número não inteiro, precisaremos obter o menor número inteiro acima deste valor. Para tal, utilizaremos uma função da biblioteca de matemática do C#, o teto, ou ***ceiling***, em inglês. Também será necessário converter o parâmetro de Ceiling() para *double*. Ainda armazenaremos o retorno do seu valor na quantidade de páginas que teremos na aplicação, e faremos um *cast* para tornar o Math.Ceiling() um int.

Depois, faremos um laço for, e então varreremos os produtos desta página em específico. Incluiremos a div neste laço e formataremos o documento com "Ctrl + K + D".

<div class = "carousel-inner" role="listbox">

@{

const int TAMANHO\_PAGINA = 4;

int paginas = (int)Math.Ceiling((double)Model.Count() / TAMANHO\_PAGINA);

}

@for (int pagina = 0; pagina < paginas; pagina++)

{

<div class="item active">...</div>

}

</div>

Assim, temos os produtos da página e, em vez de Model.Take(4), deixaremos Model.Take(TAMANHO\_PAGINA). Além disso, precisaremos pegar os produtos daquela página específica, para o qual chamaremos um método do LINQ (Consulta Integrada à Linguagem) do .NET, com Model., pulando a quantidade de elementos correspondente ao número de páginas por meio do método Skip():

<div class="item active">

<div class="row">

@{

var produtosDaPagina = Model.Skip(pagina \* TAMANHO\_PAGINA).Take(TAMANHO\_PAGINA);

}

@foreach (var produto in produtosDaPagina)

{

// código omitido

}

</div>

</div>

Rodaremos a aplicação mais uma vez e veremos todos os produtos, mas não é o que gostaríamos. Queremos exibir apenas uma página por vez, a qual ficaria rodando pelo carrossel. Vamos modificar a div de classe item active do CSS, que indica a página que está sendo exibida no momento — no caso, todas.

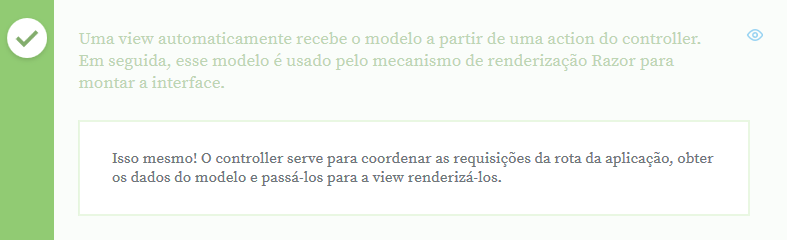
Deixaremos somente a primeira página ativa, aquela que possui índice igual a zero. Removeremos active e usaremos uma expressão C# com @ e, entre parênteses, o operador ternário para verificar se a página é igual a zero. Se sim, será retornada a *string* com o nome da classe do CSS, active e, caso contrário, o retorno será uma *string* vazia:

<div class="item @(pagina == 0 ? "active" : "")">

Vamos rodar a aplicação novamente com "Ctrl + F5". Agora sim, estamos exibindo uma página por vez e conseguimos mover o carrossel para a direita ou para a esquerda usando as setas, e acessar todos os produtos do nosso catálogo.

**Exercício 1:**

## Como uma view obtém os dados do modelo para exibir na interface do usuário?



Nesse capítulo, aprendemos:

* DataService, Injeção de Dependência
* Lendo Arquivo Json de Produtos
* Gravando Produtos no Banco de Dados
* ProdutoRepository
* Injetando Modelo na View Carrossel
* Exibindo uma Página de Produtos
* Paginação do Carrossel de Produtos

**No link a seguir, você pode fazer o** [**DOWNLOAD completo do projeto deste capítulo**](https://github.com/alura-cursos/ASPNETCore20/archive/d600b45421b0a9599455c164024fda51c66a685e.zip) **para acompanhar com o vídeo ou comparar seu código com as videoaulas.**

# Para saber mais - Code First, Database First ou Model First?

Quando trabalhamos com frameworks de "Mapeamento Objeto-Relacional", como Entity Framework, temos 3 alternativas: Code First, Database First ou Model First. Como escolher a melhor? Neste post, Gabriel Ferreira desmistifica e discute cada um deles, para você não ter dúvidas quando iniciar seu próximo projeto com Entity Framework Core.

# Qual técnica utilizar com o Entity Framework: Code First, Database First ou Model First?

**Gabriel Ferreira**

Fonte: <http://gabsferreira.com/qual-tecnica-utilizar-com-o-entity-framework-code-first-database-first-ou-model-first/>

**Aula 5 – Resumo 1:** Nossa aplicação está funcionando e, aparentemente, o catálogo está correto, mas se olharmos nossa tabela de produtos (dbo.Produto), perceberemos que os produtos são importados e duplicados no banco de dados. Isto é, o arquivo livros.json é importado repetidas vezes, cada vez que subimos o servidor.

Colocaremos um filtro no nosso repositório de produtos para fazermos uma única importação. Assim, caso o produto já exista, ele será ignorado. Vamos abrir ProdutoRepository.cs e pegar o método SaveProdutos(), no qual incluiremos o filtro para evitar a duplicação.

Caso o produto não exista, faremos a importação do mesmo, a partir do Dbset, naquele conjunto de entidades mantido pelo Entity Framework, equivalente a uma tabela e representado pelo contexto.Set<Produto>(), o qual colocaremos dentro da condição if.

Faremos uma consulta para verificar se um produto com o código que está sendo importado existe ou não, com a cláusula do LINQ Where(), que utilizará uma expressão lambda. p.Codigo terá que ser igual a livro.Codigo, e esta expressão retornará os elementos que têm o mesmo código do livro que está sendo importado.

E, então, chamaremos um método (Any()), que retornará verdadeiro ou falso, verdadeiro somente se o produto for encontrado. Caso este não seja encontrado, executaremos o código da linha que fará a importação deste produto, que moveremos para cima.

public void SaveProdutos(List<Livro> livros)

{

foreach (var livro in livros)

{

if (!contexto.Set<Produto>().Where(p => p.Codigo == livro.Codigo).Any())

{

contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));

}

}

contexto.SaveChanges();

}

Aproveitando que estamos no repositório, vamos eliminar as duplicações da expressão contexto.Set<Produto>(). Para isso, extrairemos uma **variável local** com "Ctrl + .", selecionando a opção "Introduzir local para "contexto.Set()"".

Com isso, será criado um dbSet, uma variável local que representa o DbSet de Produto. No entanto, queremos criar um campo para ser reutilizado durante o tempo de vida desta classe.

foreach (var livro in livros)

{

Microsoft.EntityFrameworkCore.DbSet<Produto> dbSet = contexto.Set<Produto>();

if (!dbSet.Where(p => p.Codigo == livro.Codigo).Any())

{

contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));

}

}

contexto.SaveChanges();

Assim, moveremos a linha para cima, e colocaremos o DbSet como campo privado:

private readonly ApplicationContext contexto;

private readonly DbSet<Produto> dbSet = contexto.Set<Produto>();

Com isto, temos a declaração do dbSet, que deixaremos só, e moveremos a atribuição do dbSet, isto é, o trecho dbSet = contexto.Set<Produto>();, para baixo, ao construtor, após o qual poderemos eliminar a atribuição da linha de cima:

private readonly ApplicationContext contexto;

private readonly DbSet<Produto> dbSet;

public ProdutoRepository(ApplicationContext contexto)

{

this.contexto = contexto;

dbSet = contexto.Set<Produto>();

}

Feito isso, substituiremos todos os lugares em que há contexto.Set<Produto>() pelo campo local dbSet, em:

public IList<Produto> GetProdutos()

{

return dbSet.ToList();

}

Assim como em:

if (!dbSet.Where(p => p.Codigo == livro.Codigo).Any())

{

dbSet.Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));

}

Em seguida, criaremos uma classe base que ajudará na eliminação destas duplicações de código, para todos os repositórios a serem utilizados. Além de ProdutoRepository, usaremos CadastroRepository, PedidoRepository e ItemPedidoRepository. A classe base será criada na pasta "Repositories" e se chamará "BaseRepository".

Criaremos esta classe copiando um trecho de código do ProdutoRepository — os dois campos locais, contexto e dbSet. Também reaproveitaremos o construtor, copiando o trecho correspondente e colando em BaseRepository.

public class BaseRepository

{

private readonly ApplicationContext contexto;

private readonly DbSet<Produto> dbSet;

public ProdutoRepository(ApplicationContext contexto)

{

this.contexto = contexto;

dbSet = contexto.Set<Produto>();

}

}

Agora, modificaremos o nome do construtor para BaseRepository, importaremos a referência de DbSet e de Produto. Nossa classe precisará ser genérica, e não específica para Produto, como está no momento. No lugar de Produto, utilizaremos um tipo genérico, um *base model*, T.

Ele terá que ser uma entidade que herdará do BaseModel, isto é, será preciso um filtro para o tipo T, com where. Também teremos que alterar a visibilidade dos campos para deixá-los como protected, pois queremos que eles sejam visíveis por todas as classes derivadas.

O código ficará da seguinte maneira:

public class BaseRepository<T> where T : BaseModel

{

protected private readonly ApplicationContext contexto;

protected private readonly DbSet<T> dbSet;

public BaseRepository(ApplicationContext contexto)

{

this.contexto = contexto;

dbSet = contexto.Set<T>();

}

}

Em ProdutoRepository.cs, colocaremos que a classe herdará de BaseRepository, cujo tipo será Produto, eliminaremos os campos locais que tínhamos criado na classe, bem como o construtor, pois iremos recriá-lo selecionando ProdutoRepository, usando "Ctrl + ." e clicando em "Gerar construtor "ProdutoRepository(contexto)"".

public class ProdutoRepository : BaseRepository<Produto>, IProdutoRepository

{

public ProdutoRepository(ApplicationContext contexto) : base(contexto)

{

}

public IList<Produto> GetProdutos()

{

return dbSet.ToList();

}

// código omitido

}

Continuando, vamos criar outras classes de repositório, selecionando a pasta "Repositories > Adicionar > Classe..." com o botão direito do mouse. Criaremos PedidoRepository, classe que será herdada de BaseRepository do tipo Pedido.

Implementaremos o construtor com "Ctrl + ." e "Gerar construtor "PedidoRepository(contexto)"". Então, faremos uma cópia deste arquivo para criarmos ItemPedidoRepository, não esquecendo de renomear a classe e o construtor no arquivo também, além de trocarmos o tipo de BaseRepository, que é ItemPedido, e não Pedido.

Por fim, faremos o mesmo para a classe CadastroRepository, ou seja, trocando a classe e o construtor para deixá-los com o mesmo nome, e substituindo Pedido por Cadastro.

Todas estas classes que criamos poderão ser injetadas por injeções de dependências. Para isto, modificaremos a classe Startup, na qual incluiremos tudo que acabamos de criar e onde se localizam as instâncias temporárias:

services.AddTransient<IDataService, DataService>();

services.AddTransient<IProdutoRepository, ProdutoRepository>();

services.AddTransient<IPedidoRepository, PedidoRepository>();

services.AddTransient<ICadastroRepository, CadastroRepository>();

services.AddTransient<IItemPedidoRepository, ItemPedidoRepository>();

Criaremos as interfaces, começando por PedidoRepository, que acessaremos e deixaremos assim:

namespace CasaDoCodigo.Repositories

{

public interface IPedidoRepository

{

}

public class PedidoRepository : BaseRepository<Pedido>, IPedidoRepository

{

public PedidoRepository(ApplicationContext contexto) : base(contexto)

}

}

Faremos o mesmo para as demais interfaces, de modo equivalente, fazendo os ajustes necessários. Estamos com os repositórios criados e prontos para serem usados!

**Resumo 2:** Vamos mexer na interface da nossa aplicação, que atualmente conta com três páginas: o carrossel, o carrinho e o cadastro do cliente. Atualmente, não estamos passando o título da página para a renderização do arquivo .cshtml. Além disso, começaremos a implementar os links utilizando ASP.NET Core MVC, já que não conseguimos navegar de uma página para a outra.

Sendo assim, abriremos nosso projeto com as quatro páginas da aplicação, Carrossel.cshtml, Carrinho.cshtml, Cadastro.cshtml e Resumo.cshtml. No carrossel, adicionaremos o título, e para isto incluiremos **@** no início do arquivo para indicar o código C#, seguida das chaves.

Daremos um nome para o título desta página, acessando um cenário de dados chamado ViewData, um dicionário de dados que é passado e renderizado pela *view*, assim como o modelo, para capturar alguma informação, no caso, o título. Colocaremos o nome Title, "título" em inglês, como chave deste dicionário. O ASP.NET Core MVC renderizará este título na aba do navegador, cujo nome será "Carrossel".

@{

ViewData["Title"] = "Carrossel";

}

Faremos o mesmo para as outras páginas, copiando e colando o trecho acima, e trocando "Carrossel" pelos títulos correspondentes, isto é, em:

* Carrinho.cshtml teremos "Carrinho";
* Cadastro.cshtml teremos "Cadastro";
* Resumo.cshtml teremos "Resumo".

Em seguida, voltaremos a Carrossel.cshtml, em que o botão "Adicionar" não realiza nenhuma ação. Teremos que criar o link para a página de carrinho, o que faremos com um elemento HMTL, a âncora a, com o href que seria o link para a página de carrinho. Em vez de utilizarmos um link com href, utilizaremos uma técnica chamada ***Tag Helper***, usada na inclusão de um atributo especial que se inicia com a palavra "asp".

Ao usarmos <a asp- class="btn btn-success">Adicionar</a>, por exemplo, o ASP.NET Core MVC nos dá uma lista de opções de atributos de *tag helper*, sendo uma delas asp-controller, que escolheremos porque queremos navegar para uma *action* de um determinado *controller*, neste caso, pedido. Também usaremos asp-action para navegarmos à página do carrinho:

<div class="text-center">

<a asp-cotroller="pedido"

asp-action="carrinho"

class="btn btn-success">Adicionar</a>

</div>

Note que, com isso, o Visual Studio mudou a cor da âncora e a deixou em negrito, indicando que o elemento a não é mais uma âncora HTML, e sim uma *tag helper*, ou seja, um componente do ASP.NET Core utilizado pelo mecanismo de renderização ***Razor*** para a criação de HTML de maneira mais natural.

A impressão é que estamos incluindo um código HTML sendo que estamos colocando um componente do servidor do ASP.NET Core. Rodaremos a aplicação novamente para vermos o resultado desta alteração. Desta vez, ao passarmos o mouse sobre o botão, o link exibido na parte inferior da tela é o endereço da página referente ao carrinho.

Adiante, implementaremos os outros links para a nossa aplicação utilizando as *tag helpers*.

**<a asp-controller="pedido"**

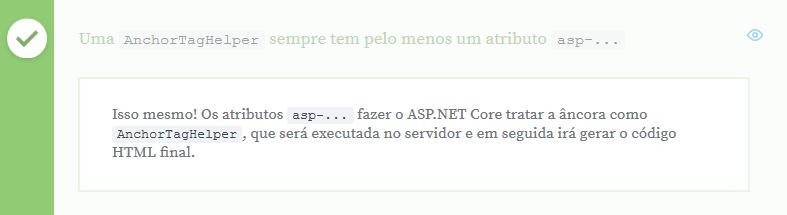
**asp-action="carrinho"**

**class="btn btn-success">Adicionar</a>**

**Não é mais uma ancora de referência ( ref=”/” ), agora é uma *tag helper***

# Âncora HTML x AnchorTagHelper

## Como o ASP.NET Core consegue diferenciar uma Âncora HTML de um AnchorTagHelper, se ambos têm o formato <a ... /a>?



**Resumo 3:** Quando utilizamos uma *tag helper* de âncora, normalmente acrescentamos a informação da *action* do *controller* também, a qual será chamada ao clicarmos no link. Felizmente, a *tag helper* possui uma conveniência: se você estiver navegando pelo mesmo *controller*, é possível omitir esta informação, e é isto que faremos aqui.

Eliminaremos o atributo asp-controller pois estamos sempre no mesmo *controller*, o **pedido**, mantendo apenas a asp-action dentro das tags <div>:

<div class="text-center">

<a asp-action="carrinho"

class="btn btn-success">Adicionar</a>

</div>

Em seguida, colocaremos o atributo asp-action nos demais links das páginas. Em Carrinho.cshtml, há o link para a adição de produtos, um elemento âncora do HTML, href="/", que transformaremos em uma *tag helper* por meio do atributo asp-action.

<div class="pull-right">

<a class="btn btn-success" asp-action="carrossel">

Adicionar Produtos

</a>

<a class="btn btn-success" asp-action="cadastro">

Preencher Cadastro

</a>

</div>

Na página Cadastro.cshtml, iremos à parte onde se encontram os botões para continuar comprando e finalizar pedido, em que também trocaremos os href="/":

<div class="form-group">

<a class="btn btn-success" asp-action="carrossel">

Continuar Comprando

</a>

</div>

Vamos rodar a aplicação para verificar seu comportamento. Clicaremos no botão "Adicionar" para incluirmos o produto no carrinho e, nesta página, poderemos pressionar "Adicionar Produtos" para voltarmos à página anterior e continuar vendo o catálogo.

Da mesma forma, conseguimos clicar no botão "Preencher Cadastro" para navegarmos à tela de cadastro. Para continuarmos comprando, existe o botão "Continuar Comprando". E, no momento de adição do produto, queremos passar a informação de código do produto que está sendo comprado à página de carrinho.

De que forma passaremos tal informação utilizando uma *tag helper*?

Acessaremos a *tag helper* de Carrossel.cshtml e, onde há asp-action, colocaremos outro atributo indicando a informação adicional, no caso, o código do produto. Porém, para passarmos qualquer informação à *tag helper*, é necessário incluir um prefixo chamado asp-route-, seguido do nome do parâmetro que estamos passando, codigo, e no final colocaremos o valor que está sendo passado, a propriedade `@produto.Codigo`:

<div class="text-center">

<a asp-action="carrinho"

asp-route-codigo="@produto.Codigo"

class="btn btn-success">Adicionar</a>

</div>

Vamos salvar, voltar ao navegador e atualizar a página. Ao passarmos o mouse sobre o botão "Adicionar" de um determinado produto, o link exibido envolve /carrinho?codigo=001, e se passarmos a outro produto, o código na URL gerada será outro. Percebemos que o código está sendo montado como uma *query string*.

É importante passarmos o código, porém existem alternativas para fazê-lo de forma mais elegante. Então, em vez de ?codigo=036, por exemplo, poderíamos ter simplesmente /036. Queremos incorporar o código do produto na rota, portanto pausaremos a aplicação e faremos uma modificação no nosso arquivo Startup.cs.

No fim do arquivo, encontraremos a definição da rota, com controller=Pedido, action=Carrossel e id, que seria o atributo passado como rota, o qual substituiremos com codigo:

routes.MapRoute(

name: "default",

template: "{controller=Pedido}/{action=Carrossel}/{codigo?}");

Vamos rodar a aplicação mais uma vez. Cairemos novamente na página de carrossel, desta vez com a mudança na hora de clicarmos nos botões de adicionar produto ao carrinho, com a URL que está sendo gerada incorporando o código do produto na rota do ASP.NET Core MVC.

Exercicio:

# Modificando a rota default

**Você está desenvolvendo uma aplicação web para consultório de odontologia.**

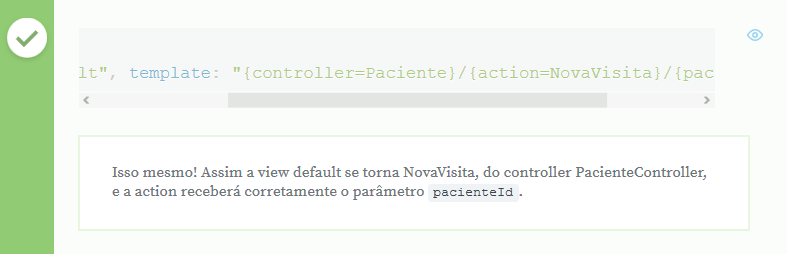
**Você precisa criar uma visualização do histórico de visitas de um paciente.**

**Você então cria um controller chamado PacienteController, onde o método com assinatura IActionResult Historico(int pacienteId) funciona como Action para a view Historico.cshtml.**

**Você quer que a view de histórico de visitas do cliente seja acessada através de uma url específica, como no exemplo abaixo:**

**[endereço do website]://paciente/historico/123**

**Onde "123" é o Id de um paciente. Além disso, você quer que a view da rota default da aplicação seja retornada pela action NovaVisita() do controller PacienteController. Como você modificaria a rota default da sua aplicação web?**



**Resumo 4:** Estamos passando o código do produto na rota para adicioná-lo ao carrinho, porém, note que a navegação passa a informação de uma página a outra. De que forma a aplicação saberá que estamos utilizando o mesmo pedido?

É importante perceber que, por definição, a web não possui estado, ou seja, é chamada de ***stateless***. Significa que cada requisição feita a uma página diferente nos traz uma nova história, e que não estamos levando a informação do pedido como deveríamos. Então, o que fazemos para manter um estado entre as páginas, para que a aplicação saiba que não importa em qual página esteja, estamos trabalhando com o mesmo pedido?

Utilizaremos uma técnica chamada **Sessão**, que mantém o estado de maneira artificial e controlada pelo servidor — estudaremos como configurá-la para mantermos o id do pedido ao longo da nossa navegação. Vamos salvar a aplicação e entrar na classe responsável pela sua configuração, Startup. Nela, encontraremos dois métodos:

* ConfigureServices(), que é onde adicionamos o serviço à aplicação;
* Configure(), em que definimos que iremos realmente utilizar o determinado serviço.

O primeiro é usado para adição, e o segundo, para configurar a sua utilização.

Em ConfigureServices(), adicionaremos o serviço de sessão AddSession(), logo após o método AddMvc(). Para fazê-lo, precisaremos incluir também outro serviço, responsável por manter as informações na memória, conforme vamos navegando, o serviço de *cache*, mais especificamente, o ***cache distribuído em memória***, por meio de AddDistributedMemoryCache():

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddDistributedMemoryCache();

services.AddSession();

// código omitido

}

Agora que configuramos o serviço neste método, vamos utilizá-lo no outro, Configure(), que expandiremos, e incluiremos:

app.UseStaticFiles();

app.UseSession();

app.UseMvc(routes =>

{

// código omitido

});

Com isso, configuramos a sessão a ser usada; entraremos no repositório de pedido e criaremos um método para obtermos o id do pedido que será armazenado na sessão. Em PedidoRepository.cs, portanto, criaremos um método privado com tipo de retorno int?, com ponto de interrogação (?) para indicar que ele pode ser nulo, seguido de GetPedidoId().

Começaremos acessando o objeto da sessão para chegarmos ao id do pedido que será armazenado, fornecido por um componente chamado de ***HTTP Context Accessor*** e, para declará-lo, poderemos criar um campo privado na classe PedidoRepository de nome contextAccessor.

public class PedidoRepository : BaseRepository<Pedido>, IPedidoRepository

{

private readonly IHttpContextAccessor contextAccessor;

public PedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

private int? GetPedidoId()

{

}

}

Em seguida, faremos a classe receber este objeto através da injeção de dependência que, por padrão, já está registrado, bastando colocá-lo como parâmetro do nosso construtor, com o qual, mais abaixo no código, faremos a atribuição do campo local. Para obtermos o pedido de id por meio de contextAccessor, colocaremos sua referência e HttpContext, após o qual acessaremos a sessão.

Então, obteremos o valor que será int, indicado por GetInt32(). Entre parênteses, incluiremos o nome da chave do dado que armazenaremos e, assim que o obtivermos, teremos que retorná-lo, então usaremos return. Para gravarmos pedidoId na sessão, criaremos outro método privado, no qual colocaremos o mesmo objeto contextAccessor:

public PedidoRepository(ApplicationContext contexto,

IHttpContextAccessor contextAccessor) : base(contexto)

{

this.contextAccessor = contextAccessor;

}

private int? GetPedidoId()

{

return contextAccessor.HttpContext.Session.GetInt32("pedidoId");

}

private void SetPedidoId(int pedidoId)

{

contextAccessor.HttpContext.Session.SetInt32("pedidoId", pedidoId);

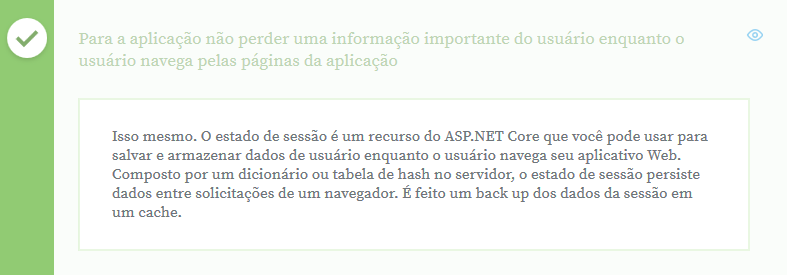
}

Com isso, definimos os dois métodos, para ler e gravar o pedido de id na sessão da nossa aplicação ASP.NET Core MVC.

**Exercicio:**

# Para quê sessão?

## **Por qual motivo você usaria a sessão (Session) numa aplicação ASP.NET Core MVC? Escolha a melhor resposta.**



Nesse capítulo, aprendemos:

* O Pattern Repository
* Links com Tag Helpers
* Passando Código de Produto na Rota
* Configurando session, Gravando e Lendo PedidoId na Sessão

**Você pode fazer o** [**DOWNLOAD completo do projeto**](https://github.com/alura-cursos/ASPNETCore20/archive/3d80a723f022766451a29ef960f0e5a1bc32162e.zip) **deste capítulo para acompanhar com o vídeo ou comparar seu código com as vídeo-aulas.**

# Para saber mais - Entity Framework Core

O **Entity Framework Core** é tão importante que a Alura Cursos Online criou um curso só para ele! Acompanhe com Daniel Portugal este novo curso para você ter mais poder sobre o banco de dados a partir da sua aplicação .NET.

## Entity Framework Core: Banco de dados de forma eficiente

**Alura Cursos Online**

**Daniel Portugal**

**Fonte:** [**https://www.alura.com.br/curso-online-entity-framework-core**](https://www.alura.com.br/curso-online-entity-framework-core)

**Aula 6 - Resumo 1:** Se estamos acessando a *action* do carrossel — o produtoRepository — para pegarmos os produtos do catálogo, de que maneira chegaremos às informações para exibirmos o carrinho na *action* correspondente?

Teremos que acessar o repositório referente aos pedidos, para o qual é necessário modificar nosso PedidoController() e criar um novo repositório como um campo privado. Faremos uma cópia deste repositório, declararemos um IPedidoRepository, passando-o em seguida por uma injeção de dependência nesta classe, adicionando um novo parâmetro no construtor.

E, então, o atribuiremos no corpo do construtor e acessaremos o repositório de pedido na *action* de carrinho, criando uma variável local chamada Pedido. Chamaremos o método para obtermos o pedido atual da sessão com GetPedido(), que ainda não existe, pois não o criamos.

O código estará assim:

public class PedidoController : Controller

{

private readonly IProdutoRepository produtoRepository;

private readonly IPedidoRepository pedidoRepository;

public PedidoController(IProdutoRepository produtoRepository,

IPedidoRepository pedidoRepository)

{

this.produtoRepository = produtoRepository;

this.pedidoRepository = pedidoRepository;

}

public IActionResult Carrossel()

{

return View(produtoRepository.GetProdutos());

}

public IActionResult Carrinho()

{

Pedido pedido = pedidoRepository.GetPedido();

return View():

}

}

Selecionaremos GetPedido, usaremos "Ctrl + ." e clicaremos na opção "Gerar método IPedidoRepository.GetPedido", com a qual o Visual Studio criará o método. Navegaremos à interface IPedidoRepository, e faremos a classe concreta PedidoRepository implementar a interface com o novo método.

Com o mouse sobre IPedidoRepository na linha public class PedidoRepository : BaseRepository<Pedido>, IPedidoRepository, usaremos "Ctrl + ." e escolheremos "Implementar interface". Teremos a interface abaixo pronta para implementação:

public Pedido GetPedido()

{

throw new NotImplementedException();

}

Como faremos para obtermos o pedido atual?

O primeiro passo é verificar qual id do pedido está gravado na sessão. Vamos declarar uma variável local com GetPedidoId(), o método que obtém a informação da sessão. Além disso, consultaremos a nossa tabela de pedidos para saber se este pedidoId existe — se não existir, criaremos um novo pedido a ser retornado.

Assim, declararemos a variável pedido, do DbSet de pedido. Criaremos um filtro com uma consulta LINQ para descobrirmos o pedido que possui o id correspondente. Na expressão lambda, usaremos p => p.Id == pedidoId, em que pedidoId se refere à sessão.

Com o método SingleOrDefault(), verificaremos se o elemento existe. Ele retornará o pedido, caso o elemento exista. Caso contrário o retorno será nulo, sem gerar erros. Na sequência, verificaremos se o pedido é nulo ou não. Se sim, será necessário criá-lo e, por este motivo, usaremos o if. Para a persistência no banco de dados, utilizaremos contexto.SaveChanges().

No caso do pedido não ser nulo, isto é, se houver alguma informação, retornaremos pedido a partir de GetPedido().

public Pedido GetPedido()

{

var pedidoId = GetPedidoId();

var pedido = dbSet.Where(p => p.Id == pedidoId)

.SingleOrDefault();

if (pedido == null)

{

pedido = new Pedido();

dbSet.Add(pedido);

contexto.SaveChanges();

}

return pedido;

}

Voltando a PedidoController.cs, passaremos a informação dos itens do pedido que acabamos de obter do repositório para a *view*. E injetaremos o parâmetro pedido.Itens no método View(), isto é, os itens do pedido obtidos do repositório:

public IActionResult Carrinho()

{

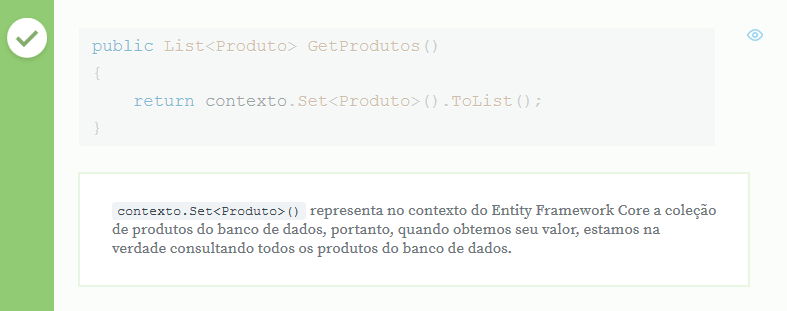
Pedido pedido = pedidoRepository.GetPedido();

return View(pedido.Itens);

}

# Exercíco: Obtendo produtos do banco de dados

## Como você implementaria o método GetProdutos() para obter todos os produtos da entidade Produto do banco de dados?



**Resumo 2:** Agora que estamos rodando a aplicação, veremos se a *action* de carrinho está conseguindo obter o novo pedido. Adicionaremos um produto qualquer, cairemos na *action* de carrinho, que por sua vez chamará o método GetPedido(). Ele criará um pedido, caso seja necessário, e chegaremos em SaveChanges(), onde o novo pedido será salvo.

Ao pressionarmos a tecla "F5", voltaremos à *action* do carrinho com o pedido criado em return View(pedido.Itens), o qual possui id 4010. Apertaremos "F5" novamente para atualizarmos a página, e o pedido será obtido mais uma vez, porém ele é recriado desnecessariamente; o certo seria mantermos o mesmo pedido por meio da sessão.

O problema é que não estamos gravando o id do pedido na sessão, portanto pausaremos a aplicação e, logo após o contexto.SaveChanges(), responsável pela gravação no banco de dados e obtenção do novo id para o pedido, pegaremos o PedidoId e o salvaremos na sessão usando o método SetPedidoId().

Com o código abaixo, gravaremos na sessão e garantimos que, a cada navegação, sempre reaproveitaremos o mesmo id.

if (pedido == null)

{

pedido = new Pedido();

dbSet.Add(pedido);

contexto.SaveChanges();

SetPedidoId(pedido.Id);

}

Vamos voltar a PedidoController.cs e modificar a *action* para passarmos o código do produto selecionado no carrossel como argumento. Adicionaremos codigo ao nosso carrinho, mas antes disto precisaremos verificar se ele foi preenchido ou não, usando uma condição:

public IActionResult Carrinho(string codigo)

{

if (!string.IsNullOrEmpty(codigo))

{

pedidoRepository.AddItem(codigo);

}

Pedido pedido = pedidoRepository.GetPedido();

return View(pedido.Itens);

}

O método AddItem() ainda não existe, então vamos criá-lo; deixaremos o mouse sobre ele, usaremos "Ctrl + ." e escolheremos "Gerar método 'IPedidoRepository.AddItem'". Em seguida, navegaremos ao novo método pressionando "F12" no teclado.

Em PedidoRepository.cs, temos a interface IPedidoRepository, e então criaremos uma implementação concreta deste método na classe PedidoRepository, clicando no nome da interface (IPedidoRepository), usando "Ctrl + ." e "Implementar interface". Então, faremos as verificações, começando pela existência ou não do código.

Declararemos uma variável de produto para obtermos o produto que possui o código correspondente no DbSet de produtos. Em seguida, faremos uma consulta com o filtro Where() e chamaremos o método SingleOrDefault() para verificarmos se existe algum produto com o código.

Caso não exista, lançaremos uma exceção, caso contrário pegaremos o pedido com uma variável local, GetPedido(), e em seguida verificaremos se o item já existe para este pedido em específico, com uma consulta simples no DbSet de ItemPedido.

Se o itemPedido não for encontrado, é necessário adicioná-lo ao carrinho. Sendo assim, criaremos outro if, com outro método para o adicionarmos. No entanto, para isto precisaremos criar uma instância de itemPedido, e no construtor passaremos como parâmetros pedido, produto, a quantidade, que inicialmente é 1, e o preço unitário, que vem de produto.Preco.

Chamaremos o contexto.SaveChanges() para podermos gravar isso no banco de dados. O código ficará da seguinte maneira:

public void AddItem(string codigo)

{

var produto = contexto.Set<Produto>()

.Where(p => p.Codigo == codigo)

.SingleOrDefault();

if (produto == null)

{

throw new ArgumentException("Produto não encontrado");

}

var pedido = GetPedido();

var itemPedido = contexto.Set<ItemPedido>()

.Where(i => i.Produto.Codigo == codigo

&& i.Pedido.Id == pedido.Id)

.SingleOrDefault();

if (itemPedido == null)

{

itemPedido = new ItemPedido(pedido, produto, 1, produto.Preco);

contexto.Set<ItemPedido>()

.Add(itemPedido);

contexto.SaveChanges();

}

}

# Exercício: Gravando id do pedido na sessão

**Você está desenvolvendo uma aplicação de comércio eletrônico, utilizando ASP.NET Core 2.0 e Entity Framework Core.**

**Num certo momento, você implementa o método GetPedido(), que obtém o Pedido atual do usuário ou cria um novo se ele não existir:**

**001 public Pedido GetPedido()**

**002 {**

**003 var pedidoId = GetPedidoId();**

**004 var pedido = dbSet**

**005 .Include(p => p.Itens)**

**006 .ThenInclude(i => i.Produto)**

**007 .Where(p => p.Id == pedidoId)**

**008 .SingleOrDefault();**

**009**

**010 if (pedido == null)**

**011 {**

**012 pedido = new Pedido();**

**013 dbSet.Add(pedido);**

**014 contexto.SaveChanges();**

**015 }**

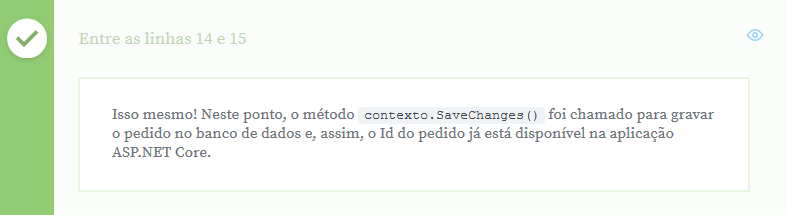
**016**

**017 return pedido;**

**018 }**

**Porém, você se esqueceu de fazer a chamada ao método SetPedidoId(int pedidoId), que grava o Id do pedido na sessão do ASP.NET Core para que ele seja preservado durante a navegação do usuário.**

**Em que lugar do código você implementaria a chamada ao método SetPedidoId?**



**Resumo 3:** Já que estamos salvando os itens de pedido, começaremos a mostrá-los na *view* de carrinho, que modificaremos, pois atualmente ela está estática e com o HTML fixo no arquivo. Abriremos Carrinho.cshtml e começaremos definindo o modelo a ser passado à *view* no início do arquivo, por meio da diretiva @model, seguido de seu tipo.

Ao voltarmos ao PedidoController.cs, veremos que chamamos uma *view* na *action* de Carrinho() passando pedido.Itens, e que os itens de pedido são uma lista de ItemPedido. Assim, modificaremos Carrinho.cshtml para passarmos como modelo um IList de ItemPedido.

@{

ViewData["Title"] = "Carrinho";

}

@model IList<ItemPedido>;

Faremos alterações para o mecanismo do Razor, que é o motor de renderização do ASP.NET Core MVC para a introdução das informações do modelo no HTML. Faremos isto identificando as partes que queremos trocar nesta *view*. Na div de classe panel-body, temos as linhas dos produtos, que iremos colapsar:

<div class="panel-body">

<div class"row row-center linha-produto">...</div>

<div class"row row-center linha-produto">...</div>

<div class"row row-center linha-produto">...</div>

</div>

Isolaremos o trecho acima e, tendo as três linhas identificadas, deixaremos somente a primeira e deletaremos as restantes. Criaremos um laço em torno deste trecho que contém somente uma linha de item de pedido para conseguirmos renderizá-la quantas vezes for necessário e montar os itens do carrinho.

Como vamos inserir um código C#, usaremos @, e uma estrutura de repetição foreach, com a div. Formataremos o código com "Ctrl + K + D".

<div class="panel-body">

@foreach (var item in Model)

{

<div class"row row-center linha-produto">...</div>

}

</div>

Precisaremos identificar as partes que estão fixas e precisam ser substituídas, como o nome da imagem de um produto, por exemplo, /images/produtos/large\_047.jpg, em que 047 indica o código. O src ficará da seguinte forma: /images/produtos/large\_@(item.Produto.Codigo).jpg.

Mais para baixo, no código, temos o nome e preço do produto:

<div class="col-md-3">Arduino Pr&#xE1;tico</div>

<div class="col-md-2 text-center">R$ 69,90</div>

Que alteraremos para:

<div class="col-md-3">@(item.Produto.Nome)</div>

<div class="col-md-2 text-center">R$ @(item.PrecoUnitario)</div>

Mais adiante, encontramos a quantidade dos itens, no caso, 2:

<input type="text" value="2"

class="form-control text center" />

<span class="input-group-btn">

<button class="btn btn-default">

<span class="glyphicon-plus"></span>

</button>

</span>

Que será substituído por @(item.Quantidade). Da mesma forma, mais para baixo há:

<div class="col-md-2">

R$ <span class="pull-right" subtotal>

139,80

</span>

</div>

Substituiremos o valor 139,80 por @(item.Quantidade \* item.PrecoUnitario), com uma operação matemática em C# porque não temos SubTotal. Adiante, temos o somatório, os totalizadores deste carrinho:

<div class="col-md-10">

<span numero-itens>

Total: 3

itens

</span>

</div>

<div class="col-md-2">

Total: R$ <span class="pull-right" total>

419,40

</span>

</div>

Em que substituiremos 3 por @(Model.Count()), e 419,40, o valor total do carrinho, por @(Model.Sum(i => i.Quantidade \* i.PrecoUnitario)), que é a soma dos subtotais dos elementos do modelo. Com isso, temos a *view* pronta para ser renderizada com os dados do modelo, portanto salvaremos o arquivo e rodaremos a aplicação no navegador.

Vamos adicionar um produto qualquer pressionando o botão "Adicionar" e seremos direcionados à tela de carrinho. Então, clicaremos em "Adicionar Produtos" para voltarmos à tela anterior e adicionarmos outro produto. Faremos isso mais uma vez, com outro livro. As telas de carrinho mostram exatamente o produto escolhido, um de cada vez, e isto é um problema...

Os produtos não são acumulados no carrinho, por conta de como a consulta obtém o pedido a ser exibido na tela de carrinho. Vamos pausar a aplicação e voltar ao PedidoRepository.cs em que encontraremos GetPedido() para a obtenção do pedido a partir do nosso DbSet, cujo código faz uma consulta em cima de uma entidade, que na verdade é convertido para uma consulta SQL do SQL Server.

Isso não é o que esperamos, pois além da consulta ao pedido, é necessário incluir as entidades que fazem parte dela. Isto é, se escrevêssemos uma consulta SQL Server, teríamos que colocar também as junções, a cláusula join para podermos juntar as tabelas.

Junto ao pedido, usaremos Include(), e resolveremos a referência com "Ctrl + .", passando uma expressão lambda, indicando a entidade a ser incluída. Além disso, acrescentaremos os produtos que o item utiliza, desta vez, não com Include(), e sim ThenInclude(), algo como "inclua em seguida de". O código, portanto, ficará da seguinte maneira:

var pedidoId = GetPedidoId();

var pedido = dbSet

.Include(p => p.Itens)

.ThenInclude(i => i.Produto)

.Where(p => p.Id == pedidoId)

.SingleOrDefault();

Vamos rodar a aplicação mais uma vez. Chegaremos à tela de carrinho, passando primeiro pela tela do carrossel. Adicionaremos um produto qualquer ao carrinho, voltaremos à tela anterior para adicionar outro, e mais um. Todos os produtos selecionados são listados na tela do carrinho, como gostaríamos.

Aprendemos como fazer uma consulta utilizando o Include() e o ThenInclude(), bem como renderizar a nossa tela de carrinho.

# Exercício:

# Consultando entidades associadas:

**Você está desenvolvendo uma aplicação web para consultório de odontologia. Sua aplicação usa ASP.NET Core e Entity Framework Core para mapear as entidades do modelo (escrito em C#), para o banco de dados relacional Sql Server.**

**As entidades do banco de dados são:**

## Paciente: O cliente que se submete ao procedimento

## Procedimento: O trabalho realizado (limpeza, extração, implante, etc.)

## Dentista: O(a) profissional de odontologia

**Suas entidades em C# possuem os seguintes relacionamentos:**

## Cada Paciente se submete a 0 ou mais procedimentos

## Cada Procedimento é realizado em 1 e somente 1 Paciente

## Cada Procedimento é realizado por 1 e somente 1 Dentista

## Cada Dentista realiza 0 ou mais procedimentos

**Para uma das consultas ao banco de dados, realizadas usando Entity Framework Core, você implementa o seguinte código para obter os dados do paciente e também os dentistas que o atenderam:**

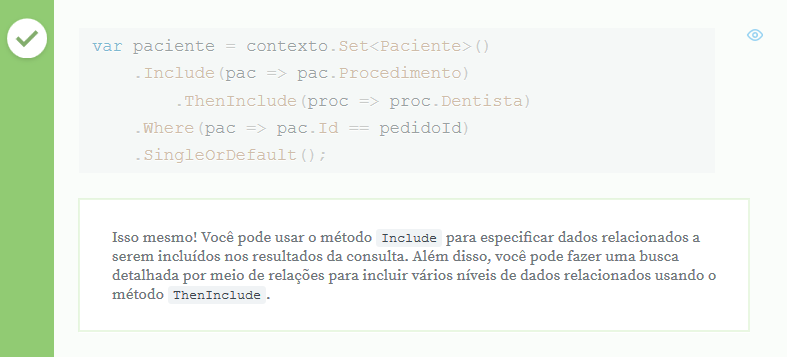
**var paciente = contexto.Set<Paciente>()**

**.Where(pac => pac.Id == pacienteId)**

**.SingleOrDefault();**

**Porém, ao obter o resultado da consulta, você percebe que o paciente contém todos os dados do paciente, porém não a consulta não trouxe nenhum Procedimento e, por causa disso, também nenhum Dentista.**

**Como você modificaria a consulta de forma a trazer também todos os procedimentos e dentistas associados ao paciente?**



**Resumo 4:** Temos a *view* de carrinho exibindo os itens que selecionamos na tela do carrossel e, pelo fluxo da aplicação, continuaremos clicando no botão "Preencher Cadastro", seguindo à tela em que preencheremos os campos com nossos dados pessoais, ao fim do qual poderemos clicar no botão "Finalizar Pedido".

Vamos exibir o resumo como se já tivéssemos preenchido os dados e finalizado o cadastro. O que faremos é uma pequena modificação na *view* do carrinho, em que temos o botão "Preencher Cadastro" que será substituído por outro, "Finalizar Pedido".

Pausaremos a aplicação e, em Carrinho.cshtml, faremos a alteração, além de substituirmos o valor de asp-action:

<div class="pull-right">

<a class="btn btn-success" asp-action="carrossel">

Adicionar Produtos

</a>

<a class="btn btn-success" asp-action="resumo">

Finalizar Pedido

</a>

</div>

Voltando à tela de navegação, iremos atualizá-la com "F5", e teremos o botão com a devida alteração. Clicaremos nele e iremos à página de resumo do pedido, da maneira como ela foi criada pela nossa webdesigner.

As informações são falsas, e os dados pessoais que ali se encontram, como e-mail, endereço de entrega, nome, telefone, itens comprados, apenas simulam uma tela deste tipo em contexto de vida real.

Começaremos a modificar a *view* de resumo da compra para exibirmos os dados do modelo dentro do nosso HTML. Acessaremos PedidoController.cs para localizar a *action* de resumo; pausaremos a aplicação e injetaremos nesta *view* o pedido a ser exibido, o qual precisará ser obtido de algum lugar.

public IActionResult Resumo()

{

Pedido pedido = pedidoRepository.GetPedido();

return View(pedido);

}

Consultaremos o nosso repositório de pedido com GetPedido() e passaremos à nossa *view* de resumo da compra. Agora que estamos passando um modelo para dentro da *view*, teremos que modificá-la para recebermos estes dados. Em Resumo.cshtml, declararemos o tipo deste modelo, incluindo a diretiva também: @model Pedido.

Feito isso, começaremos a substituir as informações que estão na página pelos dados que vêm do modelo, como o número do pedido, por exemplo, cujo 123 trocaremos por uma expressão C#:

@model Pedido;

<h3>Resumo do Pedido</h3>

<div class="panel panel-default">

<div class="panel-body">

<div class="row">

<div class="col-md-12">

<h3>Nº do Pedido: @(Model.Id)</h3>

</div>

</div>

</div>

</div>

Mais abaixo, encontramos informações pessoais como nome, telefone, e-mail e afins. Por enquanto, ignoraremos estes dados por ainda não estarmos preenchendo o cadastro. No fim do código, há outras informações, como nome do item e a quantidade correspondente.

Colapsaremos as linhas referentes aos produtos do carrinho, no caso, três, que no momento se encontram fixos na *view* e precisam ser trocados por **itens dinâmicos**. Para tal, manteremos apenas a primeira div, que deixaremos em um laço foreach para podermos varrer os itens do pedido e exibi-los na tela.

Para cada item em Model.Itens, será exibido um item dinamicamente, sendo assim substituiremos o nome do produto por `@item.Produto.Nome, e a quantidade por@item.Quantidade`. O código ficará da seguinte forma:

@foreach (var item in Model.Itens)

{

<div class="row">

<div class="col-md-10">

<div>@item.Produto.Nome</div>

</div>

<div class="col-md-2">

<div class="pull-right">@item.Quantidade</div>

</div>

</div>

}

Com isto, temos a *view* preparada. Rodaremos a aplicação novamente e criaremos um novo pedido, isto é, colocaremos os produtos no carrinho mais uma vez. Na *view* de carrossel, selecionaremos o primeiro, o segundo, o terceiro e o quarto produto. Na página de resumo do pedido, clicaremos em "Finalizar Pedido".

Na página a que somos redirecionados, temos o número do pedido, trazido do banco de dados por meio do repositório, e os quatro produtos selecionados por nós estão listados conforme esperado, com suas respectivas quantidades, que vieram do nosso modelo.

**Resumo 6:** Chegamos ao fim do curso de ASP.NET Core 2.0! O que vimos?

### Criação do projeto

Utilizando o Visual Studio, aprendemos a criar um projeto com template do MVC, padrão que envolve três componentes: o **model**, o **view** e o **controller**. Vimos como trabalhar com cada um deles de forma detalhada e modificamos a rota padrão do MVC para que pudéssemos navegar para cada uma das páginas da nossa aplicação.

### Banco de dados

Vimos como criá-lo a partir do conceito de Model First, ou Code First, para inicialmente criarmos as nossas classes em C#, após o qual foram migradas para o banco de dados SQL Server. Para isso, utilizamos o **Entity Framework Core**, que fez esta migração, ou mapeamento, para o banco de dados.

Também vimos como trabalhar com **relacionamentos** para ligar cada uma das entidades da nossa aplicação utilizando as propriedades de navegação. Além disso, lidamos com o conceito de **migração**, que são pacotes usados para pequenas atualizações do banco de dados, como tabelas, colunas, índices, e outros. São atualizações feitas inicialmente no nosso código do modelo em C#.

### Catálogo de produtos

Lidamos com o catálogo de produtos importando-os a partir de um arquivo JSON, depois convertendo este texto em dados que são gravados na tabela de produtos. Em seguida, aprendemos como exibir estes produtos na nossa tela de carrossel, de quatro em quatro, utilizando-se uma técnica de paginação feita na nossa view de carrossel com código C# e LINQ.

Por falar nisso, vimos que para combinarmos o código C# com HMTL em uma view, é possível utilizarmos um **motor de renderização** chamado **Razor**.

### Session

Trabalhamos com um padrão denominado **Pattern Repository**, que permite que cada uma das entidades do nosso modelo tenha uma classe dedicada para o acesso aos dados. Vimos o **Tag Helper**, uma série de componentes executados no servidor, que se assemelham a elementos do HTML, utilizados para gerar o código HTML de forma dinâmica e fácil.

Vimos como acessar o código do produto pelo carrossel e passar à página seguinte utilizando a rota, e como manter a informação do id do pedido ao longo da navegação do usuário por meio de **sessões**, os quais configuramos e com que conseguimos ler as informações de pedido.

### Carrinho e resumo de pedido

Trabalhamos com o carrinho e com a tela de resumo, e aprendemos a criar pedido, obtendo-o do banco de dados com o Entity Framework, e gravamos os itens selecionados na tabela, para depois os exibirmos na tela de carrinho. Por fim, aprendemos a finalizar o pedido para exibir os itens em forma de resumo na última tela da aplicação.

Espero que tenha gostado do curso, e que deixe sua avaliação para podermos melhorá-lo ainda mais. Caso haja dúvidas, consulte o [fórum](https://cursos.alura.com.br/forum/curso-webapp-com-aspnet-core-2/todos/), não deixe de participar.

Muito obrigado, até a próxima!

Nesse capítulo, aprendemos:

* Obter ou criar um pedido;
* Salvar item de pedido;
* Tela de carrinho e método Include();
* *View* de Resumo;
* Conclusão.

**Você pode fazer o** [**DOWNLOAD completo do projeto deste capítulo**](https://github.com/alura-cursos/ASPNETCore20/archive/3633c1108bd5c7d9b3cfa2333755b88bf365a39f.zip) **para acompanhar com o vídeo ou comparar seu código com as vídeoaulas.**

Depois de concluir este curso, continue aprendendo no [Curso Asp.NET Core 2.0 parte 2: Melhorando sua webapp com validações e segurança](https://cursos.alura.com.br/course/aspnet-core-2-validacoes-seguranca)!

# Curso Asp.NET Core 2.0 parte 2:

#### Melhorando sua webapp com validações e segurança.

### Faça esse curso e...

* Aprenda a organizar código JavaScript com classes e métodos
* Permita que dados do modelo trafeguem do cliente para o servidor, e vice-versa, através de Data Contract
* Atualize as quantidades dos produtos no banco de dados com Entity Framework
* Utilize Tag Helpers para agilizar a elaboração de view com formulário
* Obtenha um pedido do banco de dados, altere e atualize o pedido no banco de dados
* Reforce a segurança da aplicação web com o Token Anti-Falsificação