06

Gerando Banco, Classe Startup, Serviços, Injeção de Dependências

Transcrição

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 pip
        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 pip
        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.