



A <PROGRAMAÇÃO> PRECISA DE TI

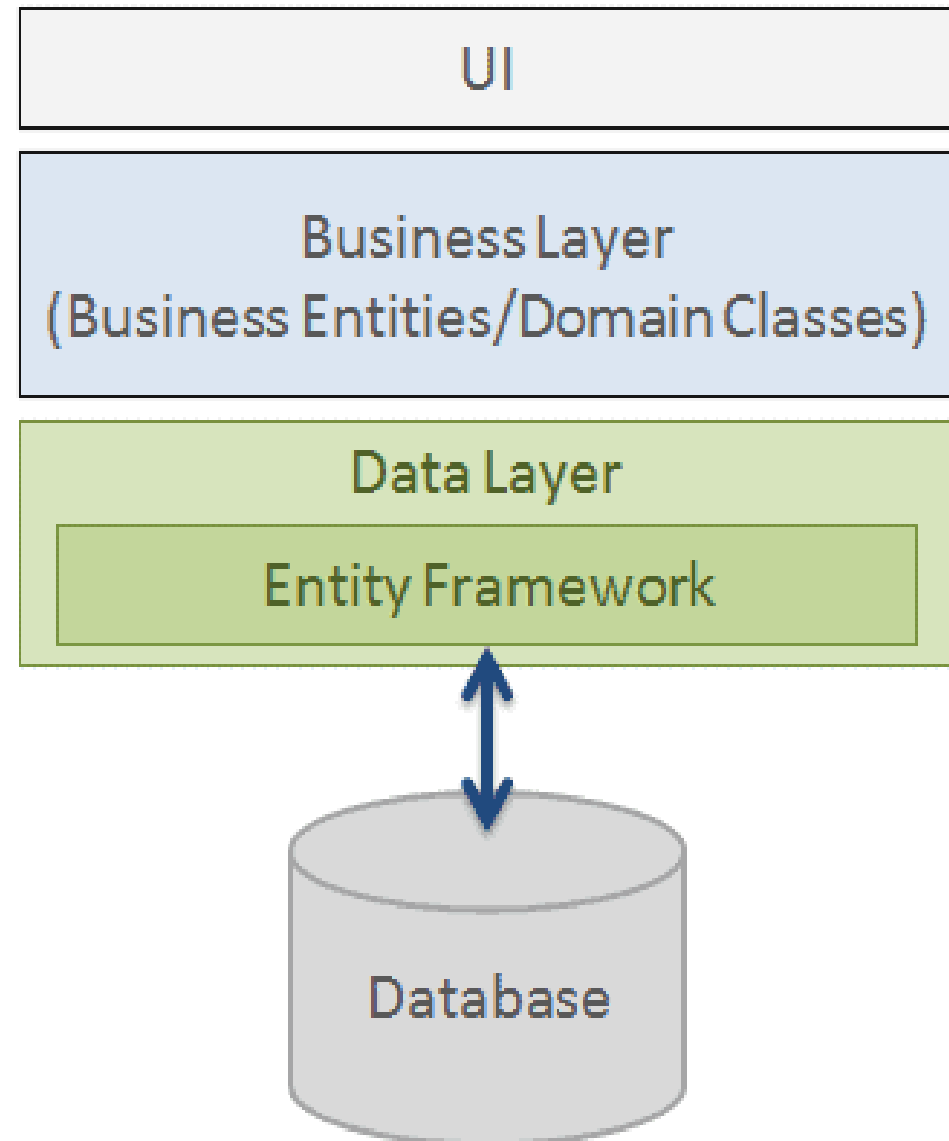
Entity Framework

Laboratório Web

Entity Framework (EF)

- É uma framework ORM (object-relational mapper) que permite interagirmos com a base de dados utilizando objetos .NET
- Introduz uma abstração na forma como lidámos com os dados
- Utilizamos objetos .NET para representar as tabelas e as colunas da BD
- Acelera a implementação todas as tarefas relacionadas com a base de dados

Onde é que
a EF
“encaixa”
nas
aplicações?



Funcionalidades - Entity Framework

- **Cross-platform:** Executável em Windows, Linux and Mac.
- **Modelling:** Utiliza modelos para representar entidades com propriedades de diferentes tipos de dados. Estes modelos são usados para efetuar operações na BD.
- **Querying:** Podemos utilizar queries LINQ (C#/VB.NET) para obter dados da BD. Estas queries são depois traduzidas para a linguagem específica da BD que estamos a utilizar. Também podemos realizar queries embutidas.
- **Change Tracking:** Temos um histórico das alterações efetuadas às instâncias das nossas entidades que tem que ser submetidas à BD.
- **Saving:** As operações INSERT, UPDATE, and DELETE só são efetuadas quando o método SaveChanges() é invocado. Também existe o método assíncrono SaveChangesAsync().

Funcionalidades - Entity Framework

- **Transactions:** Faz a gestão automática das transações para obter ou escrever dados.
- **Caching:** Suporta caching de primeiro nível por omissão. Significa que queries repetidas vão devolver dados da cache em vez da BD.
- **Built-in Conventions:** Segue convenções do padrão de desenho de configuração e inclui um conjunto de regras para configurar automaticamente o modelo EF.
- **Configurations:** Permite configurar o modelo EF através da utilização de atributos de anotação.
- **Migrations:** Providencia um conjunto de comandos de migração que podem ser executados na NuGet Package Manager Console ou na CLI para criar ou gerir o schema da BD.

Overview

Entity Framework API

Maps
Classes to
Database
Schema

Translates
LINQ-to-Entities
Queries to SQL
and executes it

Tracks
Changes

Saves
Changes to
Database

Workflow Típico - Entity Framework

- Definir o modelo
 - Inclui criar as classes de domínio
 - Criar a classe de contexto por herança da classe **DbContext**
 - Criar configurações (se necessário)
- Utilizar a classe de context para efetuar operações na BD
 - Executar queries LINQ-to-Entities para selecionar dados
 - Para adicionar dados, temos que inserir um objeto de domínio ao contexto e invocar o método *SaveChanges()*
 - Para editar ou remover dados, temos que atualizar ou remover dados do contexto e invocar o método *SaveChanges()*

Entidades – Entity Framework

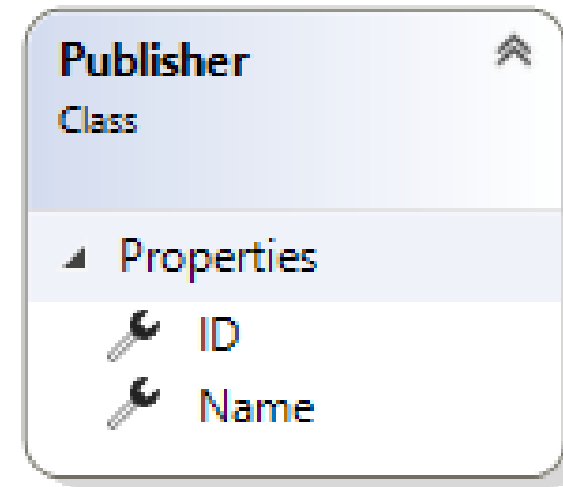
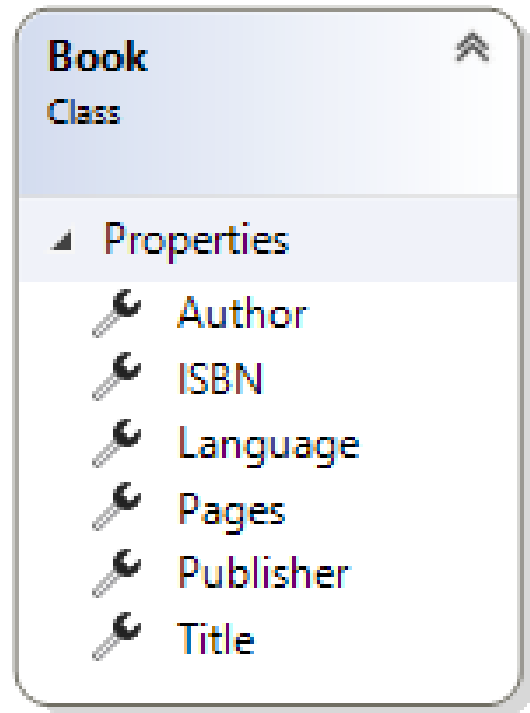
20 references

```
public class Book
{
    6 references
    public string ISBN { get; set; }
    6 references
    public string Title { get; set; }
    5 references
    public string Author { get; set; }
    5 references
    public string Language { get; set; }
    5 references
    public int Pages { get; set; }
    11 references
    public virtual Publisher Publisher { get; set; }
}
```

7 references

```
public class Publisher
{
    3 references
    public int ID { get; set; }
    3 references
    public string? Name { get; set; }
}
```


Entidades – Entity Framework



DbContext- Entity Framework

- Gestão da ligação à BD
- Configurar o modelo e as propriedades e relações das entidades
- Efetuar pesquisas na BD
- Inserir/alterar dados na BD
- Configuração do seguimento das alterações
- Caching
- Gestão das transações

Como utilizar o DbContext?

- Temos que criar uma classe que herda da classe DbContext
- Construtor por parâmetros que recebe um parâmetro do tipo `DbContextOptions<MyContext>`
- Essa classe tem que conter o seguinte:
 - Propriedades do tipo `DbSet<TEntity>` para representar conjuntos de entidades
 - Sobrecarregar o método *OnConfiguring* para efetuar selecionar e configurar a BD a ser utilizada
 - Sobrecarregar o método *OnModelCreating* para configurar o modelo (propriedades e relações das entidades/tabelas)

DbContext – Conjuntos de Entidades

7 references

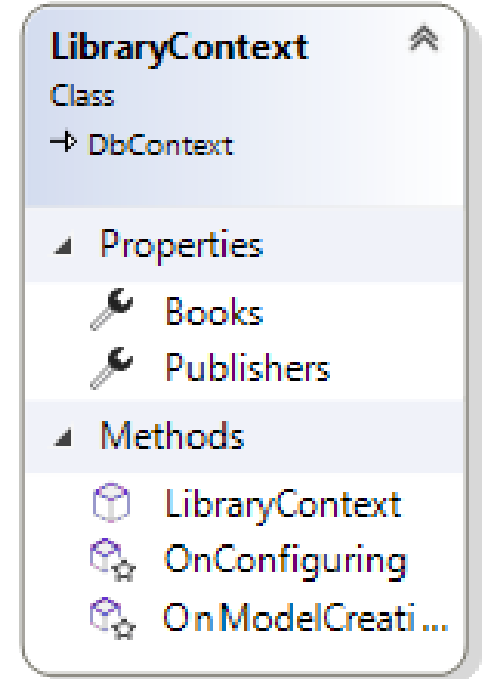
```
public class LibraryContext : DbContext
{
```

10 references

```
    public DbSet<Book> Books { get; set; }
```

5 references

```
    public DbSet<Publisher> Publishers { get; set; }
```



DbContext – Construtor e configuração

0 references

```
public LibraryContext(DbContextOptions<LibraryContext> options)
: base(options)
{
}
```

0 references

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseMySQL("server=localhost;database=library;" +
        "user=root;password=password");
}
```

DbContext – Criação do modelo

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<Publisher>(entity =>
    {
        entity.HasKey(e => e.ID);
        entity.Property(e => e.Name).IsRequired();
    });

    modelBuilder.Entity<Book>(entity =>
    {
        entity.HasKey(e => e.ISBN);
        entity.Property(e => e.Title).IsRequired();
        entity.HasOne(d => d.Publisher);
    });
}
```

DbContext – Propriedades das Entidades

- Podem ser configuradas diretamente no modelo utilizando **Data Annotations** diretamente na Entidade

```
public class Blog
{
    [Column("blog_id")]
    public int BlogId { get; set; }

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

    [NotMapped]
    public DateTime LoadedFromDatabase { get; set; }

    [Column(TypeName = "varchar(200)")]
    public string Url { get; set; }

    [MaxLength(500)]
    public string Title { get; set; }
}
```

DbContext – Para cada entidade

- Podem ser configuradas utilizando **Fluent API**, aqui não temos que modificar as classes entidades e estas configurações sobrepõem as efetuadas por Data Annotations

```
modelBuilder.Entity<Book>(entity =>
{
    entity.HasKey(e => e.ISBN);
    entity.Property(e => e.Title).IsRequired();
    entity.HasOne(d => d.Publisher);
});
```


Relações das Entidades – HasOne

```
public class SampleContext : DbContext
{
    public DbSet<Author> Authors { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Author>()
            .HasOne(a => a.Biography)
            .WithOne(b => b.Author);
    }
}

public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public AuthorBiography Biography { get; set; }
}

public class AuthorBiography
{
    public int AuthorBiographyId { get; set; }
    public string Biography { get; set; }
    public int AuthorId { get; set; }
    public Author Author { get; set; }
}
```

Relações das Entidades – HasMany

```
public class Company
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<Employee> Employees { get; set; }
}

public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Company Company { get; set; }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Company>()
        .HasMany(c => c.Employees)
        .WithOne(e => e.Company);
}
```

Serviços

- Vamos criar interface e uma classe de serviço que define as operações CRUD para uma determinada entidade
- Essas operações serão realizadas através de um atributo da classe que herda da DbContext
- Este serviço será utilizado pelo controller para uma determinada entidade/recurso
- O serviço será registado na aplicação utilizando Dependency Injection

Serviços – Interface

```
public interface IBookService
{
    2 references
    public abstract IEnumerable<Book> GetAll();

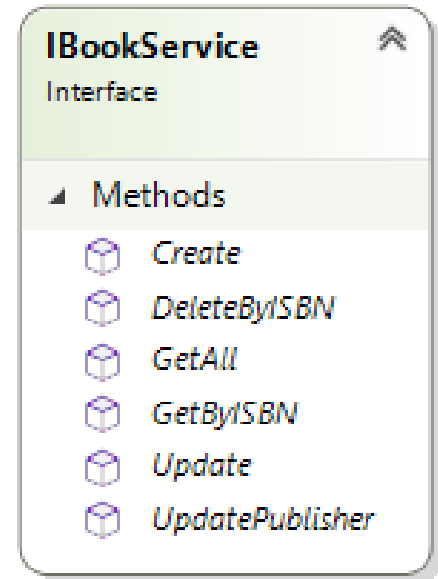
    5 references
    public abstract Book GetByISBN(string isbn);

    2 references
    public abstract Book Create(Book newBook);

    2 references
    public abstract void DeleteByISBN(string isbn);

    2 references
    public abstract void Update(string isbn, Book book);

    2 references
    public abstract void UpdatePublisher(string isbn, int publisherId);
}
```



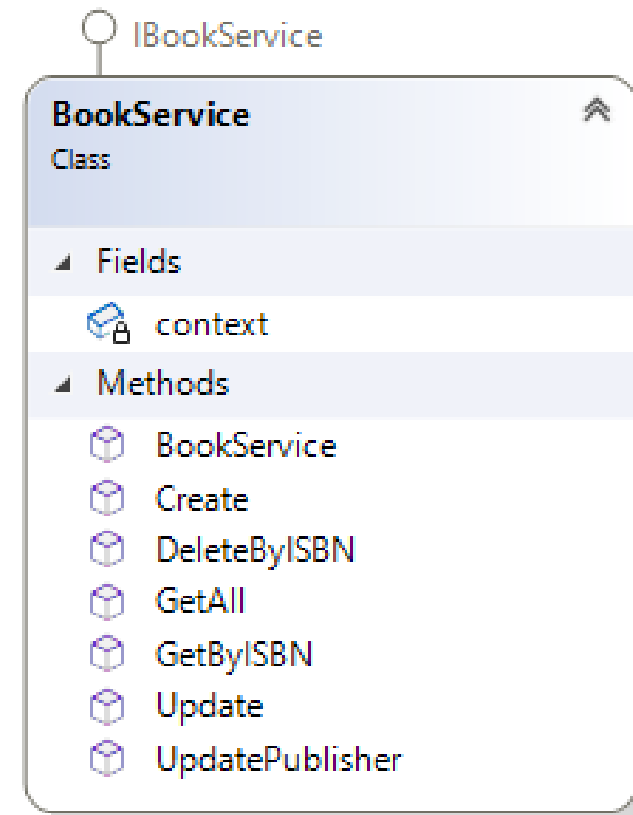
Serviços – Classe

```
public class BookService : IBookService
{
    private readonly LibraryContext context;

    0 references
    public BookService(LibraryContext context)
    {
        this.context = context;
    }

    2 references
    public IEnumerable<Book> GetAll()
    {
        var books = context.Books
            .Include(p => p.Publisher);
        return books;
    }

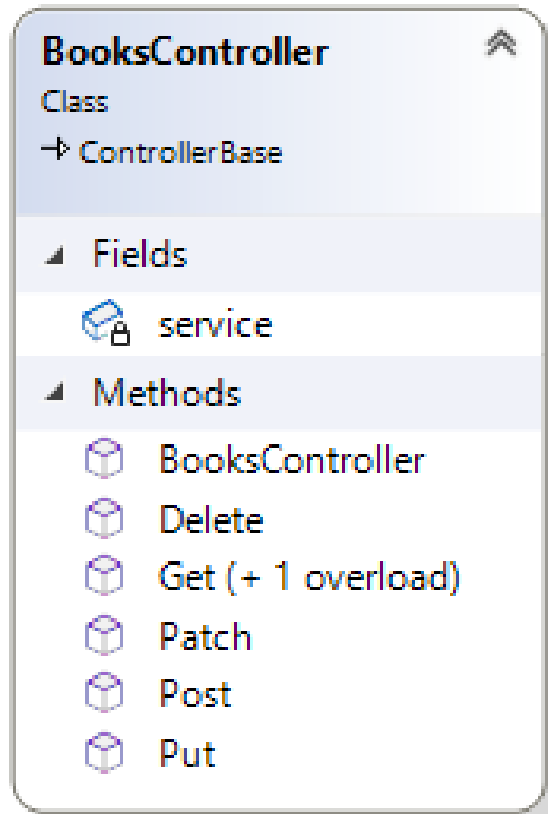
    5 references
    public Book GetByISBN(string isbn)
    {
        var book = context.Books.Include(b => b.Publisher).SingleOrDefault(b => b.ISBN == isbn);
        return book;
    }
}
```



Controller

- O controller terá um atributo do tipo da interface do serviço
- O controller vai expôr os endpoints HTTP para os verbos GET, POST, PUT, DELETE
- Estes verbos HTTP vão invocar os métodos CRUD correspondentes que foram implementados no serviço
- O serviço será injetado no construtor do controller utilizando DI

Controller



```
public class BooksController : ControllerBase
{
    private readonly IBookService service;

    0 references
    public BooksController(IBookService service)
    {
        this.service = service;
    }

    // GET: api/<BooksController>
    [HttpGet]
    0 references
    public IEnumerable<Book> Get()
    {
        return service.GetAll();
    }
}
```

Criar a BD e Inserir dados?

- Podemos criar a BD de raiz através de código .NET
- Da mesma forma que criamos uma extensão para criar um middleware personalizado, aqui vamos usar uma extensão para criar a BD
- Essa extensão vai utilizar o contexto que foi criado por nós para ler o modelo e criar a BD utilizando o método **context.Database.EnsureCreated()**
- Podemos inserir dados logo após a criação da base de dados

Criar a BD e Inserir dados?

```
public static void CreateDbIfNotExists(this IHost host)
{
    {
        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;
            var context = services.GetRequiredService<LibraryContext>();
            // Creates the database if not exists
            if (context.Database.EnsureCreated())
            {
                LibraryDbInitializer.InsertData(context);
            }
        }
    }
}
```

março de 22

Criar a BD e
inserir os
dados

```
public static class LibraryDbInitializer
{
    1 reference
    public static void InsertData(LibraryContext context)
    {
        // Adds a publisher
        var publisher = new Publisher
        {
            Name = "Mariner Books"
        };
        context.Publishers.Add(publisher);

        // Adds some books
        context.Books.Add(new Book
        {
            ISBN = "978-0544003415",
            Title = "The Lord of the Rings",
            Author = "J.R.R. Tolkien",
            Language = "English",
            Pages = 1216,
            Publisher = publisher
        });
    }
}
```



Adicionar:

- **contexto** para a BD
 - **serviço** com tempo de vida **scoped**
 - **extensão** para criar a BD
-

```
builder.Services.AddDbContext<LibraryContext>();  
builder.Services.AddScoped<IBookService, BookService>();
```

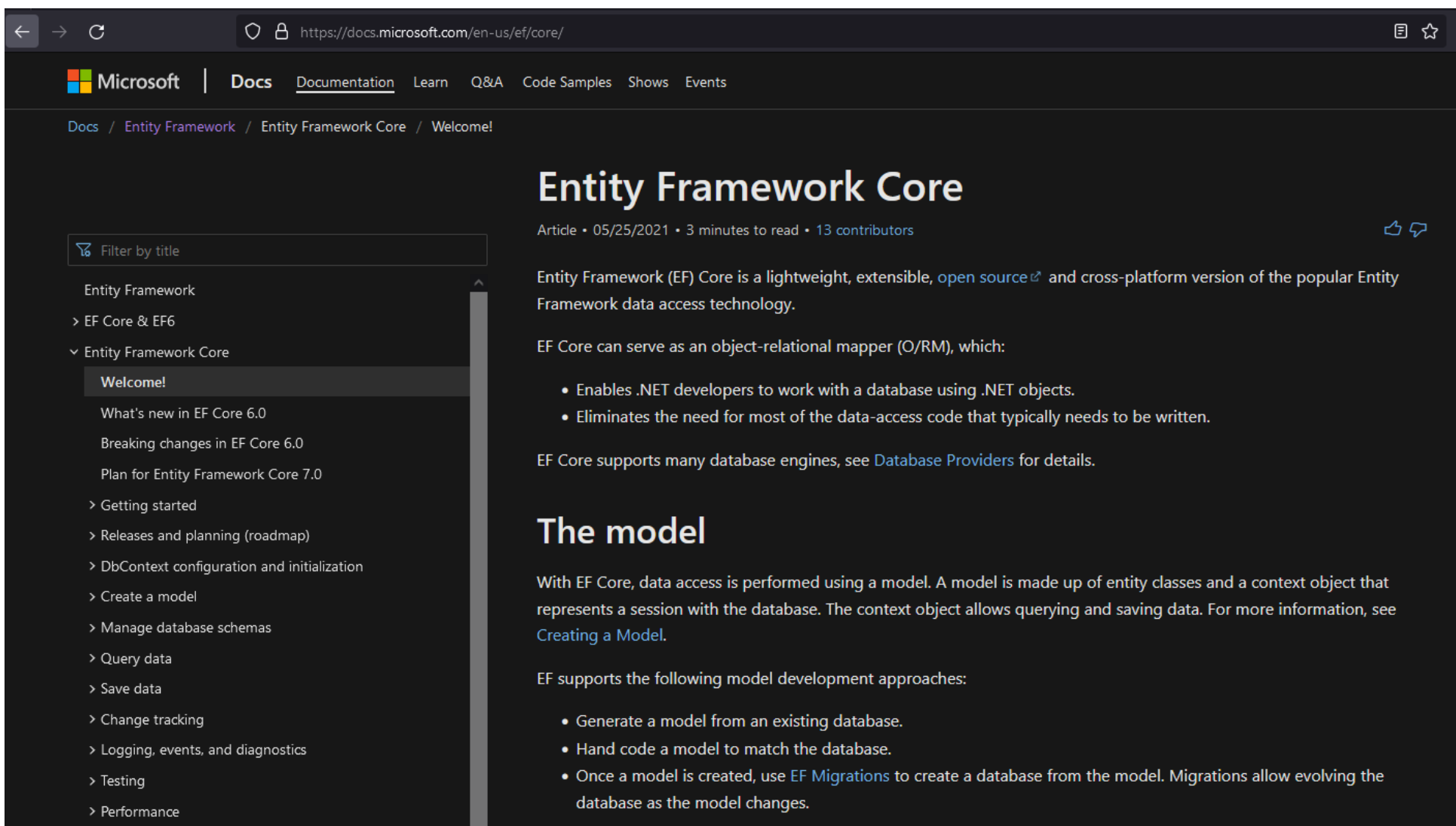
```
var app = builder.Build();  
// Configure the HTTP request pipeline.  
if (app.Environment.IsDevelopment())  
{  
    app.UseSwagger();  
    app.UseSwaggerUI();  
}
```

```
app.UseHttpsRedirection();  
app.UseAuthorization();  
app.MapControllers();
```

```
app.CreateDbIfNotExists();
```

```
app.Run();
```

Referências



The screenshot shows the Microsoft Docs website for Entity Framework Core. The browser address bar displays <https://docs.microsoft.com/en-us/ef/core/>. The navigation bar includes the Microsoft logo and links to Docs, Documentation, Learn, Q&A, Code Samples, Shows, and Events. The breadcrumb trail reads: Docs / Entity Framework / Entity Framework Core / Welcome!.

Entity Framework Core

Article • 05/25/2021 • 3 minutes to read • 13 contributors

Entity Framework (EF) Core is a lightweight, extensible, [open source](#) and cross-platform version of the popular Entity Framework data access technology.

EF Core can serve as an object-relational mapper (O/RM), which:

- Enables .NET developers to work with a database using .NET objects.
- Eliminates the need for most of the data-access code that typically needs to be written.

EF Core supports many database engines, see [Database Providers](#) for details.

The model

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. For more information, see [Creating a Model](#).

EF supports the following model development approaches:

- Generate a model from an existing database.
- Hand code a model to match the database.
- Once a model is created, use [EF Migrations](#) to create a database from the model. Migrations allow evolving the database as the model changes.

Left sidebar navigation:

- Filter by title
- Entity Framework
 - > EF Core & EF6
 - ▼ Entity Framework Core
 - Welcome!**
 - What's new in EF Core 6.0
 - Breaking changes in EF Core 6.0
 - Plan for Entity Framework Core 7.0
 - > Getting started
 - > Releases and planning (roadmap)
 - > DbContext configuration and initialization
 - > Create a model
 - > Manage database schemas
 - > Query data
 - > Save data
 - > Change tracking
 - > Logging, events, and diagnostics
 - > Testing
 - > Performance

qualificar

TAL

X

</>

A <PROGRAMAÇÃO