Entity Framework Core

Session #04

dotnet training

Agenda

Entity Framework Core

Getting Started

DbContext

Model

Query Data

Save Data

Migrations

Scaffolding

Getting Started

Entity Framework Core

Getting Started

- 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 (ORM), 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 vs. EF6

Getting Started

- Entity Framework Core (EF Core) is a modern object-database mapper for .NET. Now on version 7.0
- Entity Framework 6 (EF6) is an object-relational mapper designed for .NET Framework but with support for .NET Core.
- EF6 is a stable, supported product, but is no longer being actively developed.
- More details on comparison here: https://learn.microsoft.com/en-us/ef/efcore-and-ef6/#feature-comparison

Entity Framework Core

Features

- EF Core supports many database engines, like SQL Server, PostgreSQL, MySQL, …
- EF Core supports LINQ (Language-Integrated Query), which allows developers to write expressive and efficient queries using C#
- EF Core also provides a variety of performance-related features such as lazy loading, change tracking, and caching

Entity Framework Core

Features

- **Code first**: EF Core allows developers to create a database from code, which enables a more agile and test-driven development workflow.
- **Migrations**: EF Core has built-in support for creating and managing database migrations, which allows for easy management of database changes over time.
- **Performance improvements**: EF Core has been optimized for performance and can handle large datasets efficiently
- **Strongly Typed**: You define your database schema using C# classes known as entities, these classes are then mapped to the corresponding tables in the database, then the compiler can catch any errors related to the schema at compile-time, rather than at runtime.

EF Core flow

Flow

1. Install Entity Framework Core libraries using the NuGet package
   - `Microsoft.EntityFrameworkCore.Tools`
   - `Microsoft.EntityFrameworkCore.SqlServer`
2. Create model, using code-first or database-first approach
3. Create, read, update & delete on your app using LINQ to make it easier to express queries

DbContext

DBContext

Definition

- `DbContext` class represents a session with a database and provides an API for communicating with the database with the following capabilities:
  - Database Connections
  - Data operations such as querying and persistence
  - Change Tracking
  - Model building
  - Data Mapping
  - Object caching
  - Transaction management

DBContext

Lifecycle

- Creation of a DbContext instance
- Tracking of entity instances by the context. Entities become tracked by
  - Being returned from a query -> more common
  - Being added or attached to the context
- Changes are made to the tracked entities as needed to implement the business rule
- SaveChanges or SaveChangesAsync is called. EF Core detects the changes made and writes them to the database.
- The DbContext instance is disposed

DBContext

Considerations

- It is very important to **dispose the DbContext after use**. This ensures both that any unmanaged resources are freed, and that any events or other hooks are unregistered so as to prevent memory leaks in case the instance remains referenced.

- **DbContext is not thread-safe**. Do not share contexts between threads. Make sure to await all async calls before continuing to use the context instance.

- An `InvalidOperationException` thrown by EF Core code **can put the context into an unrecoverable state**. Such exceptions indicate a program error and are not designed to be recovered from.

DBContext

Dependency Injection

- ASP .NET Core have a specific dependency injection handler for DBContext

```
services.AddDbContext<ApplicationDbContext>(
        options => options.UseSqlServer("name=ConnectionStrings:DefaultConnection"));
```

DBContext

Database Provider

Database system	Example configuration	NuGet package
SQL Server or Azure SQL	.UseSqlServer(connectionString)	Microsoft.EntityFrameworkCore.SqlServer
Azure Cosmos DB	.UseCosmos(connectionString, databaseName)	Microsoft.EntityFrameworkCore.Cosmos
SQLite	.UseSqlite(connectionString)	Microsoft.EntityFrameworkCore.Sqlite
EF Core in-memory database	.UseInMemoryDatabase(databaseName)	Microsoft.EntityFrameworkCore.InMemory
PostgreSQL*	.UseNpgsql(connectionString)	Npgsql.EntityFrameworkCore.PostgreSQL
MySQL/MariaDB*	.UseMySql(connectionString)	Pomelo.EntityFrameworkCore.MySql
Oracle*	.UseOracle(connectionString)	Oracle.EntityFrameworkCore

DBSet<TEntity>

DBContext

- `DbSet<TEntity>` class represents a collection for a given entity within the model and is the gateway to database operations against an entity
- `DbSet<TEntity>` classes are added as properties to the DbContext
- Each property is mapped by default to database tables that take the name of the `DbSet<TEntity>` property.
- The DbSet is an implementation of the Repository pattern

DBSet<TEntity>

DBContext

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Model

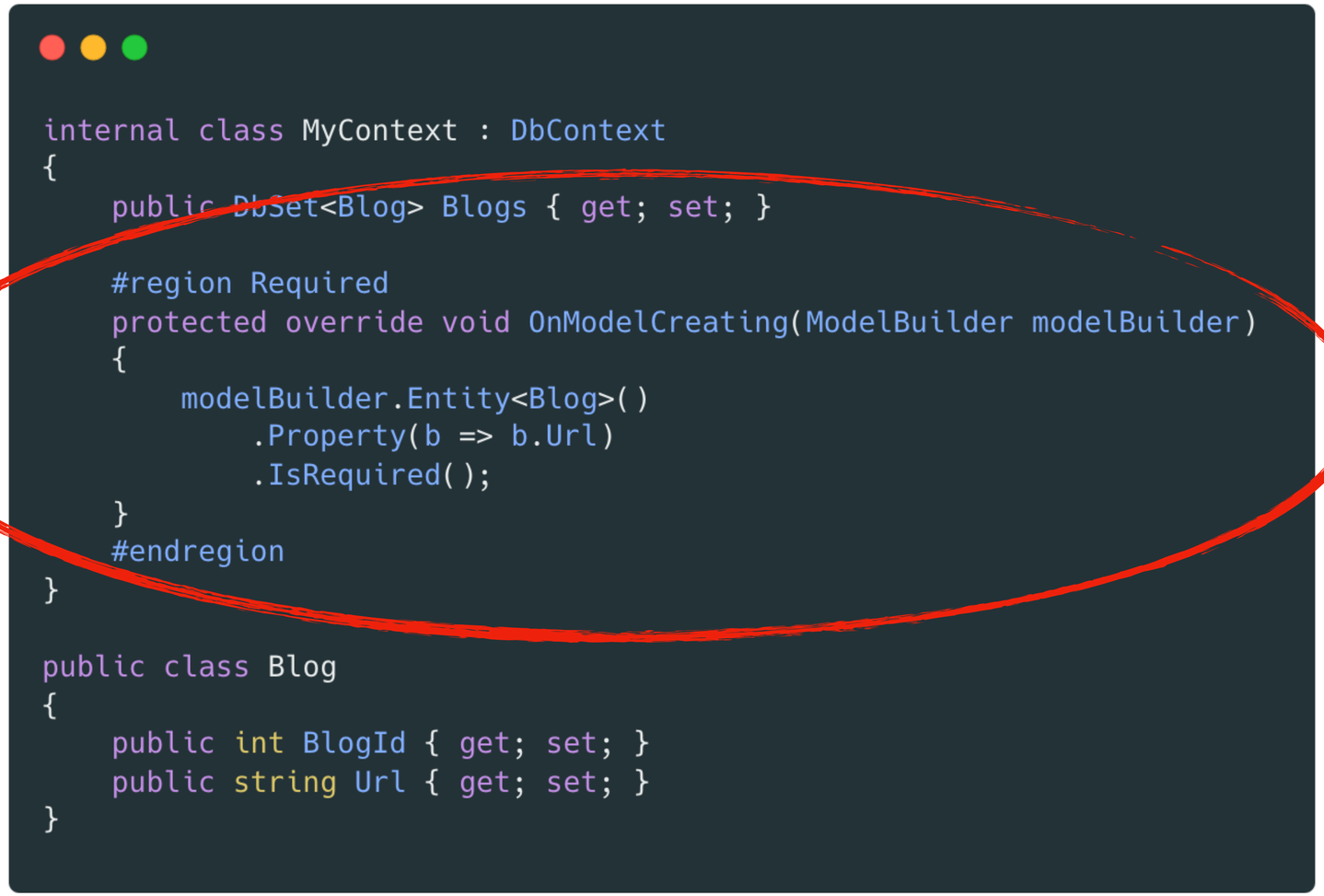
Model

How to create?

- You can use code-first or database-first approach
- More used is code-first that allow you to put object-oriented concepts in better shape that with database-first approach
- Two available ways to configure: Fluent API or Data Annotations
- Fluent API is more powerful that Data Annotations making the recommended way yo configure
- EF Core have several conventions like property names and types

Fluent API

Model



```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    #region Required
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .IsRequired();
    }
    #endregion
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Annotations

Model

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
}

[Table("Blogs")]
public class Blog
{
    public int BlogId { get; set; }

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

Entities

Model

- Types that are exposed in DbSet properties on your context are included in the model as entities
- Entity types that are specified in the OnModelCreating method are also included
- Types that are found by recursively exploring the navigation properties of other discovered entity types are included either

Entities

Model

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AuditEntry>();
    }
}
```

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public Blog Blog { get; set; }
}

public class AuditEntry
{
    public int AuditEntryId { get; set; }
    public string Username { get; set; }
    public string Action { get; set; }
}
```

Entities: Table Schema

Model

```
[Table("blogs", Schema = "blogging")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("blogging");
}
```

Entity Properties

Conventions

- SQL Server maps DateTime properties to datetime2(7) columns
- String properties to nvarchar(max) columns (or to nvarchar(450) for properties that are used as a key)
- Uses NRT (nullable reference types) to define is a property is required or nullable
- You can still use Data Annotations or Fluent API to specify that directly on the code

Entity Properties

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>(
        eb =>
        {
            eb.Property(b => b.BlogId).HasColumnName("blog_id");
            eb.Property(b => b.Url).HasColumnType("varchar(200)");
            eb.Property(b => b.Rating).HasColumnType("decimal(5, 2)");
        });
}
```

Entity Properties

Data Annotation

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

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

    [Column(TypeName = "decimal(5, 2)")]
    public decimal Rating { get; set; }
}
```

Entity Properties

Using NRT

```
public class Customer
{
    public int Id { get; set; }
    public string FirstName { get; set; } // Required by convention
    public string LastName { get; set; } // Required by convention
    public string? MiddleName { get; set; } // Optional by convention

    // Note the following use of constructor binding, which avoids compiled warnings
    // for uninitialized non-nullable properties.
    public Customer(string firstName, string lastName, string? middleName = null)
    {
        FirstName = firstName;
        LastName = lastName;
        MiddleName = middleName;
    }
}
```

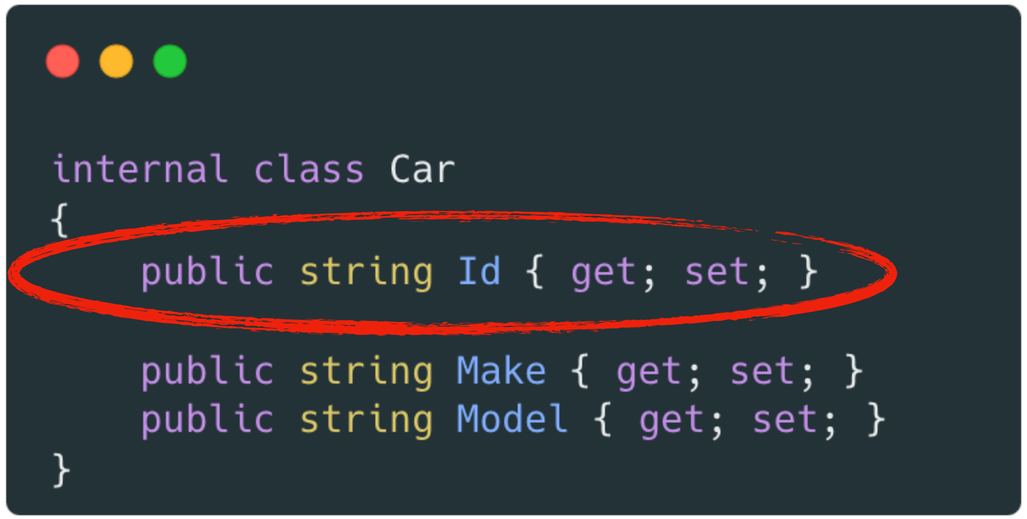
Entity Keys

Conventions

- By convention, a property named `Id` or `<type name>Id` will be configured as the primary key of an entity
- For non-composite numeric and GUID primary keys, EF Core sets up value generation for you by convention
- For example, a numeric primary key in SQL Server is automatically set up to be an IDENTITY column
- Composite keys can be configured using Data Annotation or Fluent API

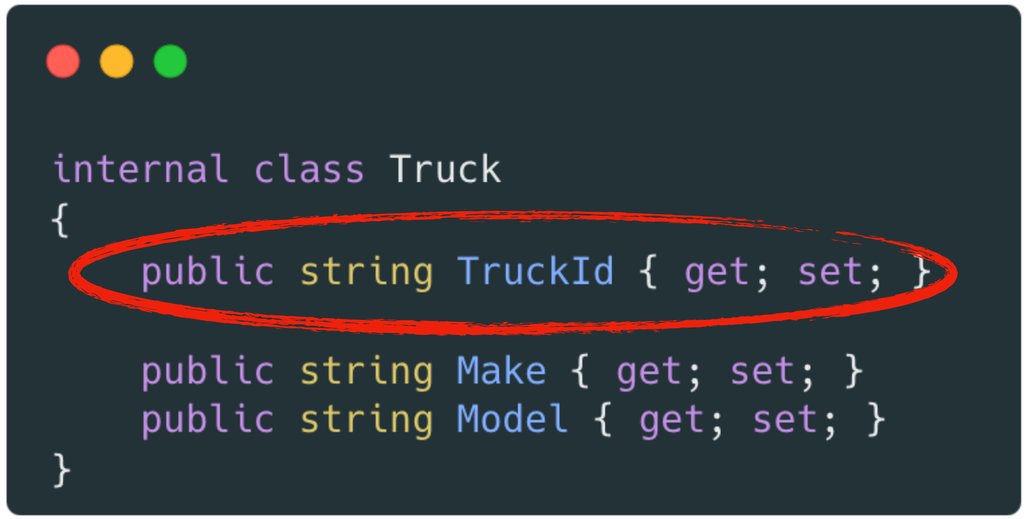
Entity Keys

Simple Keys



```
internal class Car
{
    public string Id { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```



```
internal class Truck
{
    public string TruckId { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Entity Keys

Composite Keys

```
[PrimaryKey(nameof(State), nameof(LicensePlate))]
internal class Car
{
    public string State { get; set; }
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Entity Keys

Simple Keys

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasKey(c => new { c.State, c.LicensePlate });
}
```

Generated Values

Default values

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Rating)
        .HasDefaultValue(3);
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Created)
        .HasDefaultValueSql("getdate()");
}
```

Generated Values

Generate on Add and Update

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Inserted)
        .ValueGeneratedOnAdd();
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.LastUpdated)
        .ValueGeneratedOnAddOrUpdate();
}
```

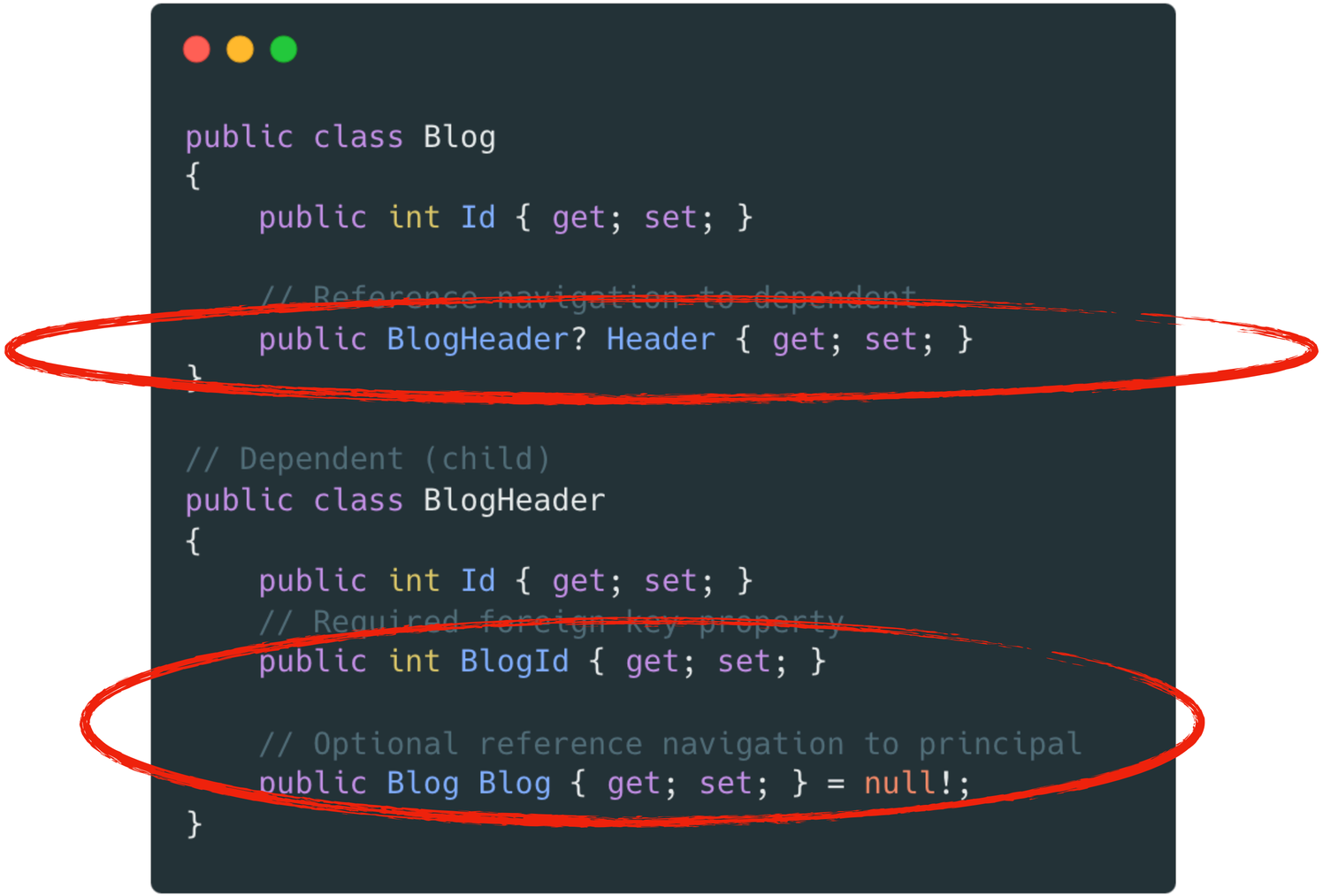
Entity Relationship

One-to-one

- One **primary key property** on the principal entity
- One **foreign key property** on the dependent entity
- Optionally, a **collection navigation** on the principal entity referencing the dependent entity
- Optionally, a **reference navigation** on the dependent entity referencing the principal entity

Entities Relationships

One-to-One



```
public class Blog
{
    public int Id { get; set; }

    // Reference navigation to dependent
    public BlogHeader? Header { get; set; }
}

// Dependent (child)
public class BlogHeader
{
    public int Id { get; set; }
    // Required foreign key property
    public int BlogId { get; set; }

    // Optional reference navigation to principal
    public Blog Blog { get; set; } = null!;
}
```

Entity Relationship

One-to-Many

- One **primary key property** on the principal entity; that is the "one" end of the relationship.
- One **foreign key property** on the dependent entity; that is the "many" end of the relationship
- Optionally, a **collection navigation** on the principal entity referencing the dependent entities
- Optionally, a **reference navigation** on the dependent entity referencing the principal entity

Entities Relationships

One-to-many

```
// Principal (parent)
public class Blog
{
    public int Id { get; set; }
    // Collection navigation containing dependents
    public ICollection<Post> Posts { get; } = new List<Post>();
}

// Dependent (child)
public class Post
{
    public int Id { get; set; }
    // Required foreign key property
    public int BlogId { get; set; }
    // Required reference navigation to principal
    public Blog Blog { get; set; } = null!;
}
```

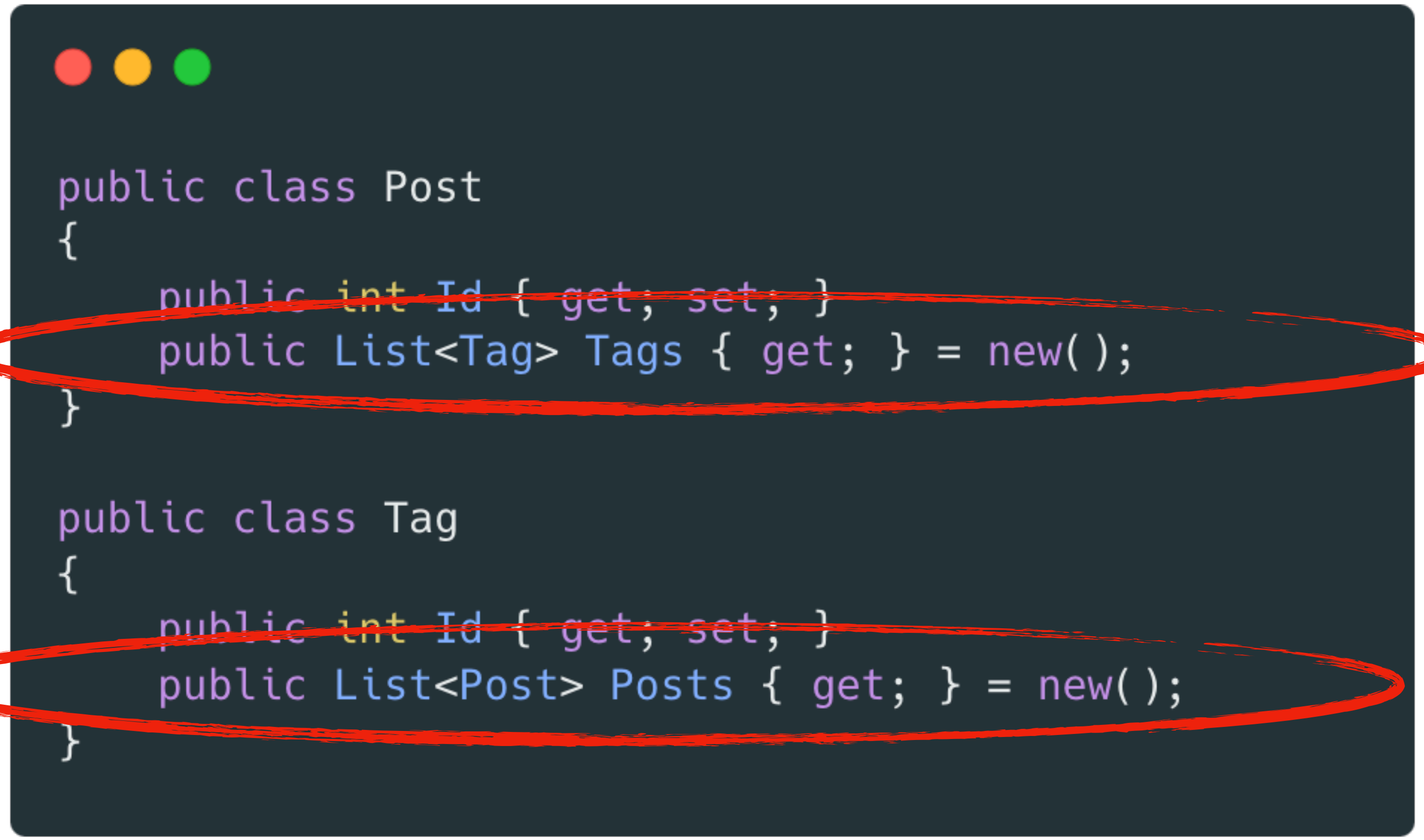
Entity Relationship

Many-to-Many

- Can be created with several approaches
- EF Core make this mapping simple using only two collection navigation on each side of the relationship
- Other option is to use a join table
- Additional options here: https://learn.microsoft.com/en-us/ef/core/modeling/relationships/many-to-many

Entity Relationship

Many-to-Many



```
public class Post
{
    public int Id { get; set; }
    public List<Tag> Tags { get; } = new();
}

public class Tag
{
    public int Id { get; set; }
    public List<Post> Posts { get; } = new();
}
```

Entity Relationship

Many-to-Many

```
public class Post
{
    public int Id { get; set; }
    public List<Tag> Tags { get; } = new();
}

public class Tag
{
    public int Id { get; set; }
    public List<Post> Posts { get; } = new();
}

public class PostTag
{
    public int PostId { get; set; }
    public int TagId { get; set; }
}
```

Query Data

Loading All Data

Query Data

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.ToList();
}
```

Loading Single Entity

Query Data

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);
}
```

Filtering

Query Data

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
}
```

Server vs. Client Execution

Query Data

```
var blogs = context.Blogs
    .OrderByDescending(blog => blog.Rating)
    .Select(
        blog => new { Id = blog.BlogId, Url = StandardizeUrl(blog.Url) })
    .ToList();
```

Load Related Data

Query Data

- By default, uses the “no load” option
- **Eager loading** means that the related data is loaded from the database as part of the initial query.
- **Explicit loading** means that the related data is explicitly loaded from the database at a later time.
- **Lazy loading** means that the related data is transparently loaded from the database when the navigation property is accessed
  - This option is not natively available and need an external library and code changes (https://learn.microsoft.com/en-us/ef/core/querying/related-data/lazy)

Eager Loading

Load Related Data

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ToList();
}
```

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .ToList();
}
```

Explicit Loading

Load Related Data

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    context.Entry(blog)
        .Collection(b => b.Posts)
        .Load();

    context.Entry(blog)
        .Reference(b => b.Owner)
        .Load();
}
```

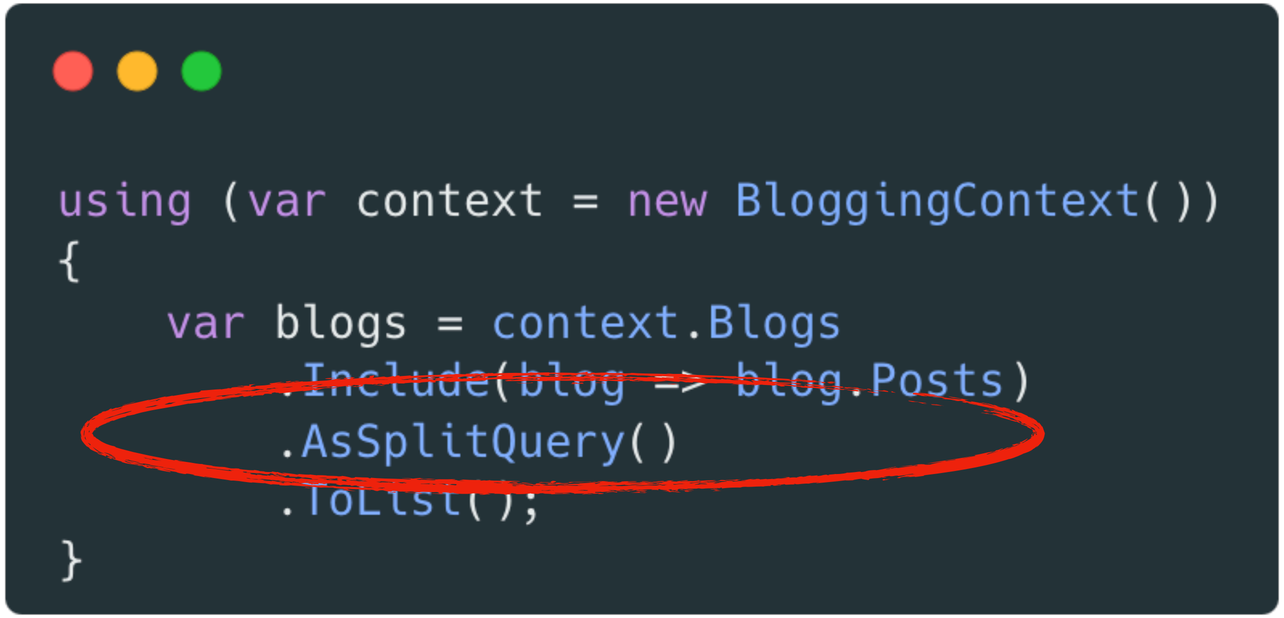
Load Related Data

Query Data

- When working against relational databases, EF loads related entities by introducing JOINs into a single query
- While JOINs are quite standard when using SQL, they can create significant performance issues if used improperly
- To avoid that you may split your queries (or use lazy load)

Split Queries

Load Related Data



Always Use Split Queries

Load Related Data

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=EFQuerying",
            o => o.UseQuerySplittingBehavior(QuerySplittingBehavior.SplitQuery))
}
```

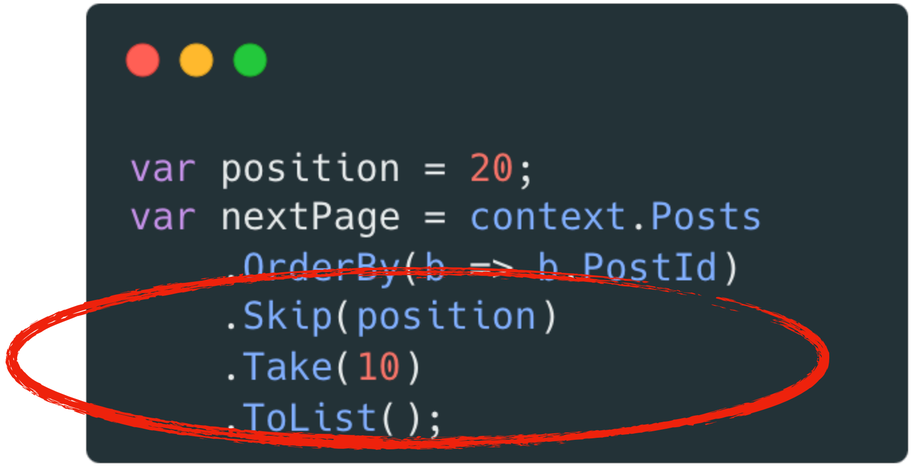
Pagination

Query Data

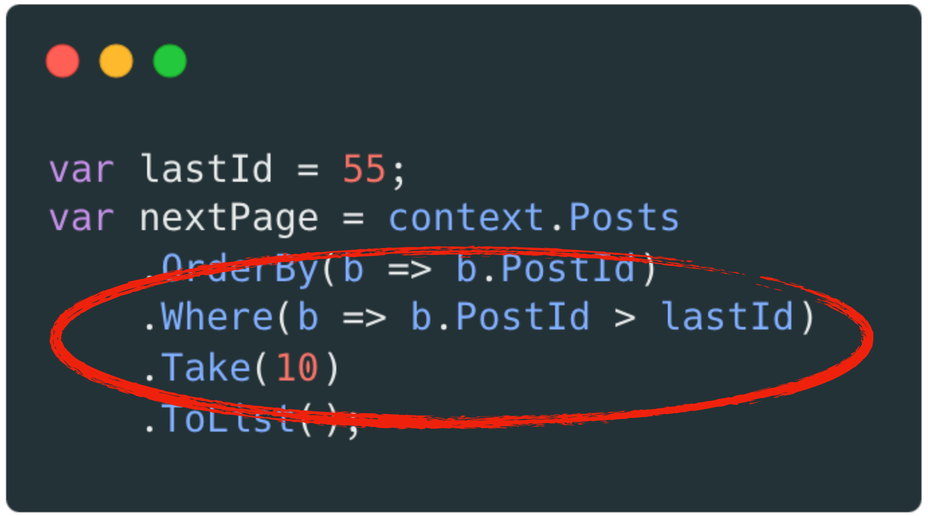
- Pagination refers to retrieving results in pages, rather than all at once
- This is typically done for large resultset and allows to navigate to the next or previous page of the results
- Offset pagination, using Skip and Take
  - The database must still process the skipped entries
  - If any updates occur concurrently, your pagination may end up skipping certain entries or showing them twice.
- Keyset pagination, use a `where` clause to skip rows, instead of an offset

Offset Pagination vs Keyset Pagination

Query Data



```
var position = 20;
var nextPage = context.Posts
    .OrderBy(b => b.PostId)
    .Skip(position)
    .Take(10)
    .ToList();
```



```
var lastId = 55;
var nextPage = context.Posts
    .OrderBy(b => b.PostId)
    .Where(b => b.PostId > lastId)
    .Take(10)
    .ToList();
```

Save Data

Save data

Query Data

- Saving data means adding new entities to the database, removing entities, or modifying the properties of existing entities in some way
- Change tracking and SaveChanges
  - Query data from database using DBContext
  - Make your changes
  - SaveChanges when your done
  - Execute inside a DB Transaction
- ExecuteUpdate and ExecuteDelete (bulk update)
  - Specify rows to be affected
  - Execute the action in all of them

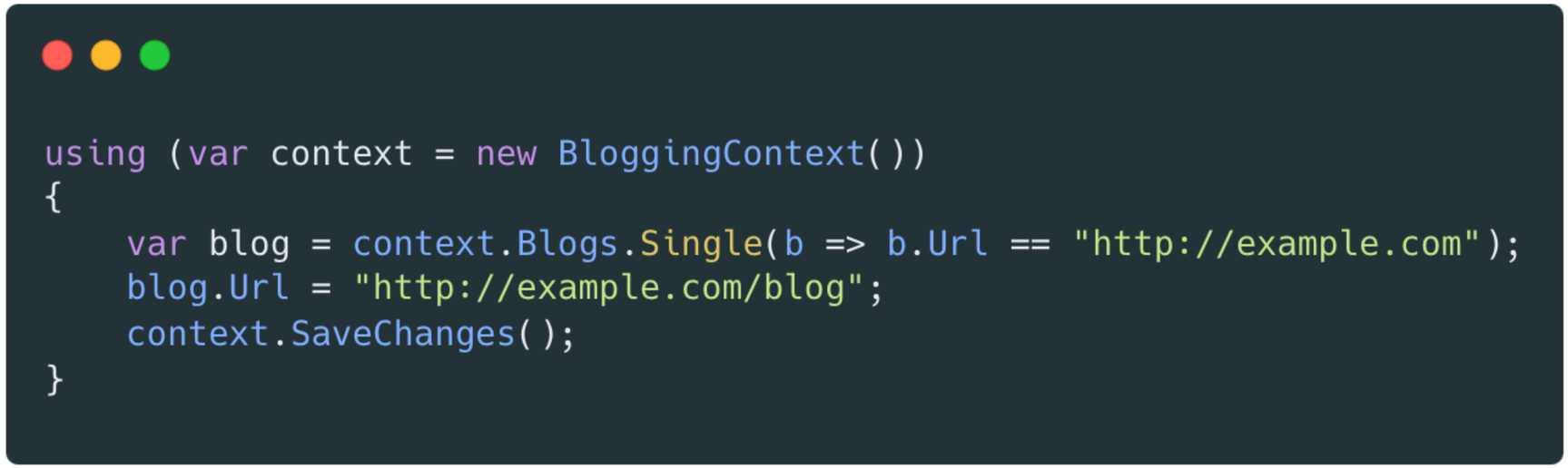
Adding Data

Basic Save

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Url = "http://example.com" };
    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

Updating Data

Basic Save



```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Single(b => b.Url == "http://example.com");
    blog.Url = "http://example.com/blog";
    context.SaveChanges();
}
```

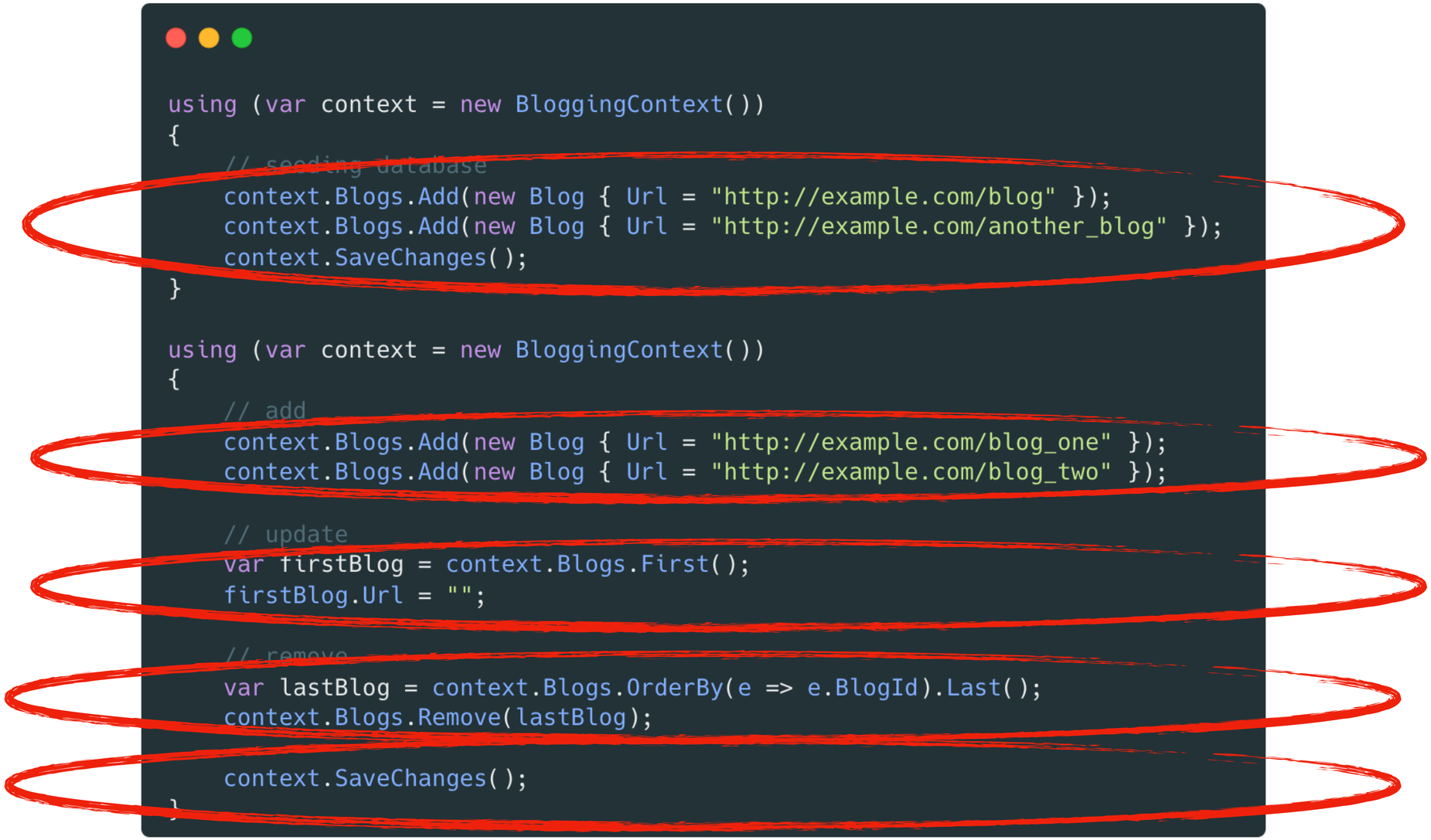
Remove Data

Basic Save

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Single(b => b.Url == "http://example.com/blog");
    context.Blogs.Remove(blog);
    context.SaveChanges();
}
```

Multiple Changes on Single SaveChanges

Basic Save



```
using (var context = new BloggingContext())
{
    // seeding database
    context.Blogs.Add(new Blog { Url = "http://example.com/blog" });
    context.Blogs.Add(new Blog { Url = "http://example.com/another_blog" });
    context.SaveChanges();
}

using (var context = new BloggingContext())
{
    // add
    context.Blogs.Add(new Blog { Url = "http://example.com/blog_one" });
    context.Blogs.Add(new Blog { Url = "http://example.com/blog_two" });

    // update
    var firstBlog = context.Blogs.First();
    firstBlog.Url = "";

    // remove
    var lastBlog = context.Blogs.OrderBy(e => e.BlogId).Last();
    context.Blogs.Remove(lastBlog);

    context.SaveChanges();
}
```

Related Data

Basic Save

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Include(b => b.Posts).First();
    var post = new Post { Title = "Intro to EF Core" };

    blog.Posts.Add(post);
    context.SaveChanges();
}
```

ExecuteDelete

Bulk Update

```
foreach (var blog in context.Blogs.Where(b => b.Rating < 3))
{
    context.Blogs.Remove(blog);
}

context.SaveChanges();
```

ExecuteUpdate

Bulk Update

```
context.Blogs
    .Where(b => b.Rating < 3)
    .ExecuteUpdate(setters => setters.SetProperty(b => b.IsVisible, false));
```

Migrations

Migrations

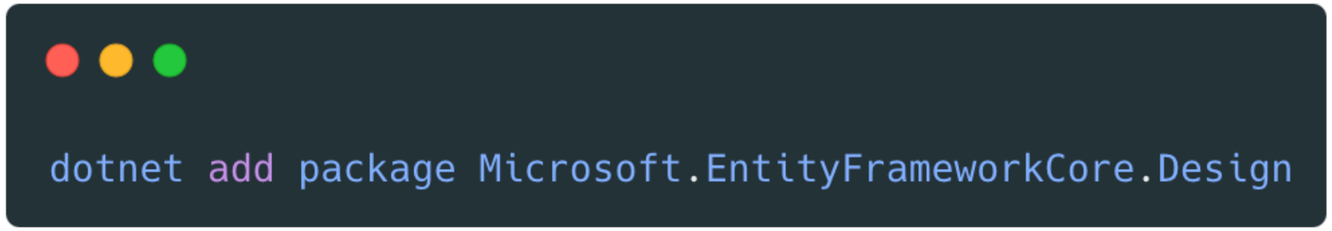
Concept

- When a data model change is introduced, the developer uses EF Core tools to add a corresponding migration describing the updates necessary to keep the database schema in sync
- EF Core compares the current model against a snapshot of the old model to determine the differences, and generates migration source files
- The files can be tracked in your project's source control like any other source file
- Once a new migration has been generated, it can be applied to a database in various ways
- EF Core records all applied migrations in a special history table, allowing it to know which migrations have been applied and which haven't

Migrations
Dotnet CLI tools

- To enable this you need to instal a dotnet CLI tool

- And then use a package on your code



Create your first migration and database

Migrations

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

Evolve your model

Migrations

```
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime CreatedTimestamp { get; set; }
}
```

Generete SQL

Migration

- If you prefer (or need) you can generate the SQL script to be applied on your database

```
dotnet ef migrations script <from_migration> <to_migration>
```

- If you already evolve your data, you may create a idempotent script

```
dotnet ef migrations script --idempotent
```

Scaffolding

Reverse Engineering

Scaffolding

- Reverse engineering is the process of scaffolding entity type classes and a DbContext class based on a database schema
- It can be performed using the `dotnet ef dbcontext` scaffold command of the .NET Command-line Interface (CLI) tools
- On Visual Studio, you may use the `Scaffold-DbContext` command of the EF Core Package Manager Console (PMC) tools
- If you use Visual Studio, EF Core Power Tools community extension provide a graphical tool which builds on top of the EF Core command line tools and offers additional workflow and customization options

Generate Model

Scaffolding

```
dotnet ef dbcontext scaffold "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Chinook" Microsoft.EntityFrameworkCore.SqlServer
```

Generate Model with ConnectionString

Scaffolding

```
{
  "ConnectionStrings": {
    "Chinook": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=Chinook"
  }
}
```

```
dotnet ef dbcontext scaffold "Name=ConnectionStrings:Chinook" Microsoft.EntityFrameworkCore.SqlServer
```

Using secrets

Scaffolding

- You can use the connection string defined on your app settings.json file
- But this can make you have sensitive information available on plain text
- Dotnet CLI provide a simple way to define secrets that are stored encrypted on your machine

Using secrets

Scaffolding

```
dotnet user-secrets init
```

```
dotnet user-secrets set ConnectionStrings:Chinook "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Chinook"
```

```
dotnet ef dbcontext scaffold "Name=ConnectionStrings:Chinook" Microsoft.EntityFrameworkCore.SqlServer
```

Q&A

Connect your API to SQL database

Lab #04

dotnet training

Connect your API to SQL database

Lab #04

- Learning Objectives
  - Create your SQL Server database with code-first approach
  - Use Entity Framework Core to connect to database
  - Manage migrations
  - Use dependency injection to inject your DbContext
  - Connect your API to a SQL Server

- MD Link: https://github.com/tasb/dotnet-training/blob/main/labs/lab04.md
- HTML Link: https://tasb.github.io/dotnet-training/labs/lab04.html