Развој на серверски WEB апликации

Entity Framework Core

Data in MVC applications

- Web applications often use information and they usually require a data store for that information.
- The data store is usually a database, but other types of data stores are occasionally used.
- In Model-View-Controller (MVC) applications, you can create a model that implements data access logic and business logic.
- When you write an ASP.NET core application you can use the Entity Framework Core (EF Core) and Language Integrated Query (LINQ) technologies, which make data access code very quick to write and simple to understand.

Object Relational Mapper (ORM)

- Developers write code that works with classes and objects.
- In contrast, databases store data in tables with columns and rows, and database administrators create and analyze databases by running Transact-SQL queries
- ORM is a programming technique that simplifies the application's interaction with data by using a metadata descriptor to connect object code to a relational database.
- ORM provides an abstraction that maps application objects to database records.
- Entity Framework is an ORM framework created by Microsoft and it maps the tables and columns found in a database to the objects and properties that are used in ASP.NET code.

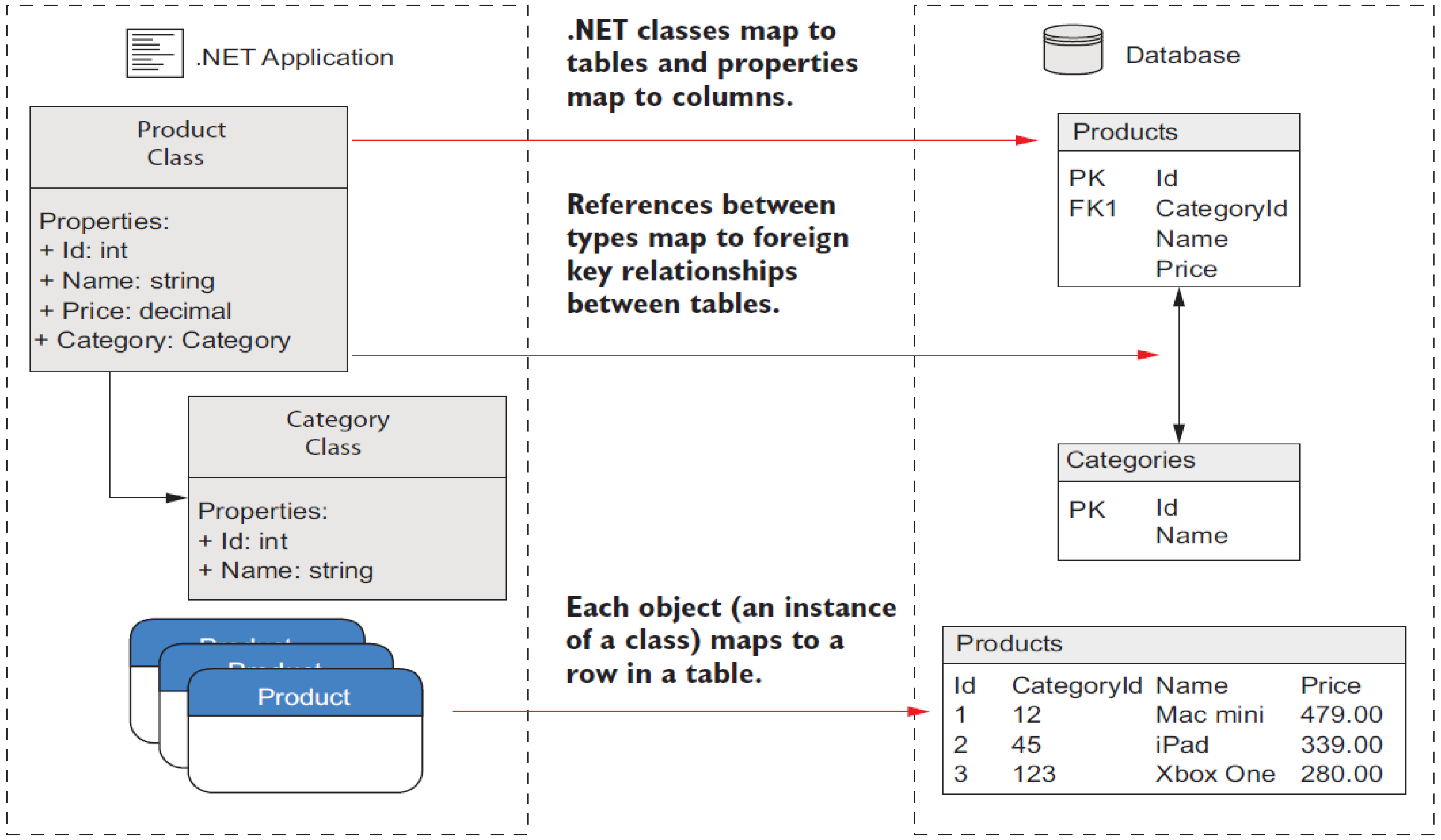
Overview of Entity Framework

- Entity Framework provides a one-stop solution to interact with data that is stored in a database.
- Instead of writing plain-text SQL statements, you can work with your own domain classes, i.e., entities, and you do not have to parse the results from a tabular structure to an object structure.
- Entity Framework keeps track of the changes you make to the entities to enable updating the database with data that exists in memory.
- Entity Framework introduces an abstraction layer between the database schema and the code of your application, which makes your application more flexible.

Entity Framework Core Intro

- Entity Framework (EF) Core is the .NET Core version of Entity Framework.
- EF Core is an object-relational mapping (ORM) framework that can be used when developing ASP.NET Core applications.
- It allows developers to work with databases and use the data in ASP.NET Core applications.
- Moreover, it enables developers to be data-oriented without the need to concentrate on modeling the entities.
- EF Core is a light-weight version of Entity Framework. In addition, it is also extensible and cross-platform.

EF Core maps .NET classes to database tables



Products

Id	CategoryId	Name	Price
1	12	Mac mini	479.00
2	45	iPad	339.00
3	123	Xbox One	280.00

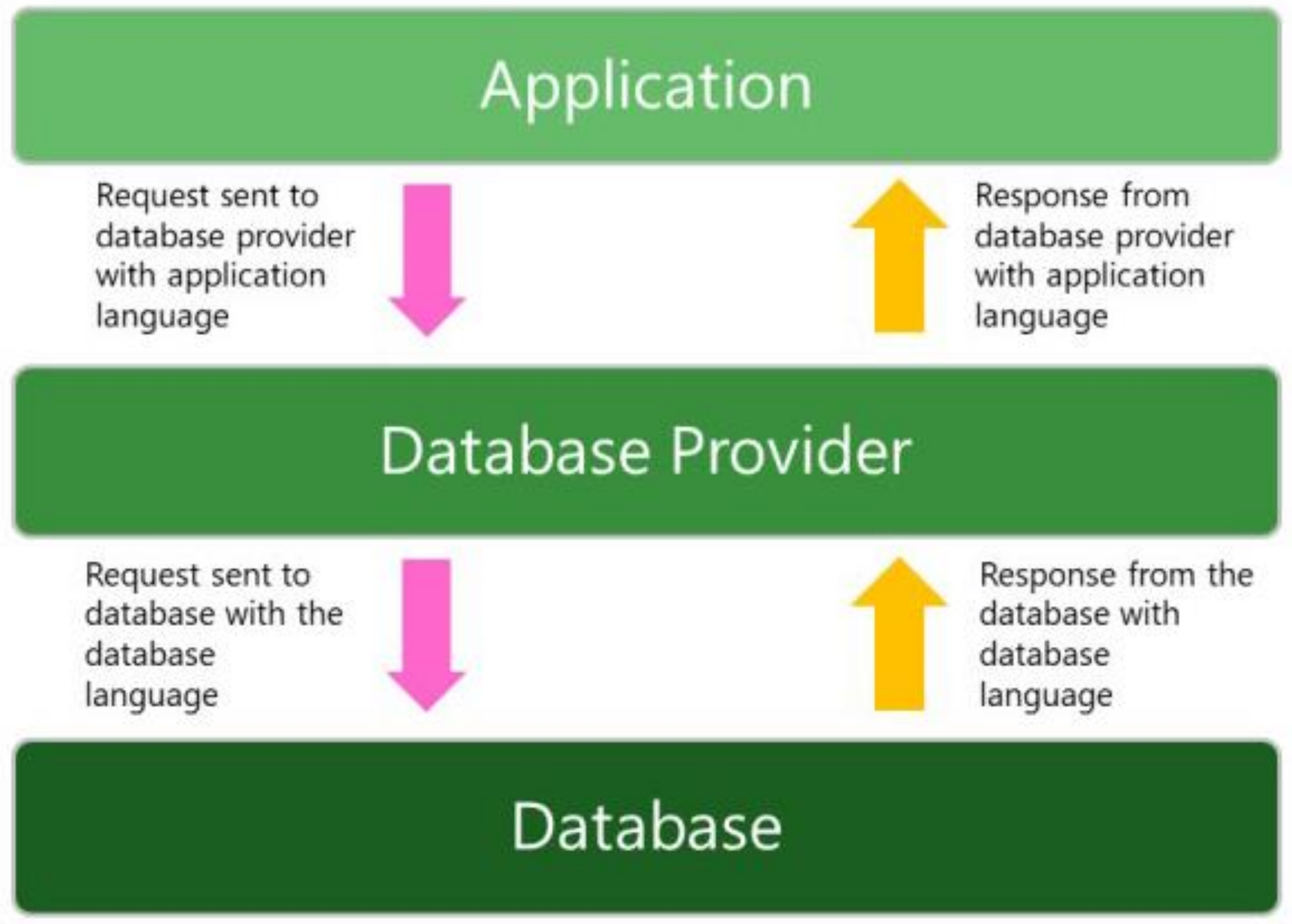
Entity Framework Core Approaches

- Entity Framework Core provides two general approaches to create your data access layer (DAL) of the application:
  - Creating entity & context classes for an existing database is called **Database-First approach** → if your database existed prior to creating the data model, you can Scaffold-DbContext to reverse engineer the data model from the database tables.
  - **Code First** is a technique which helps us to create a database, migrate and maintain the database and its tables from the code → Entity Framework Core scans your domain classes and their properties and tries to map them to the database.
- Code First is preferred in EF Core – following slides focus solely on the code first approach.

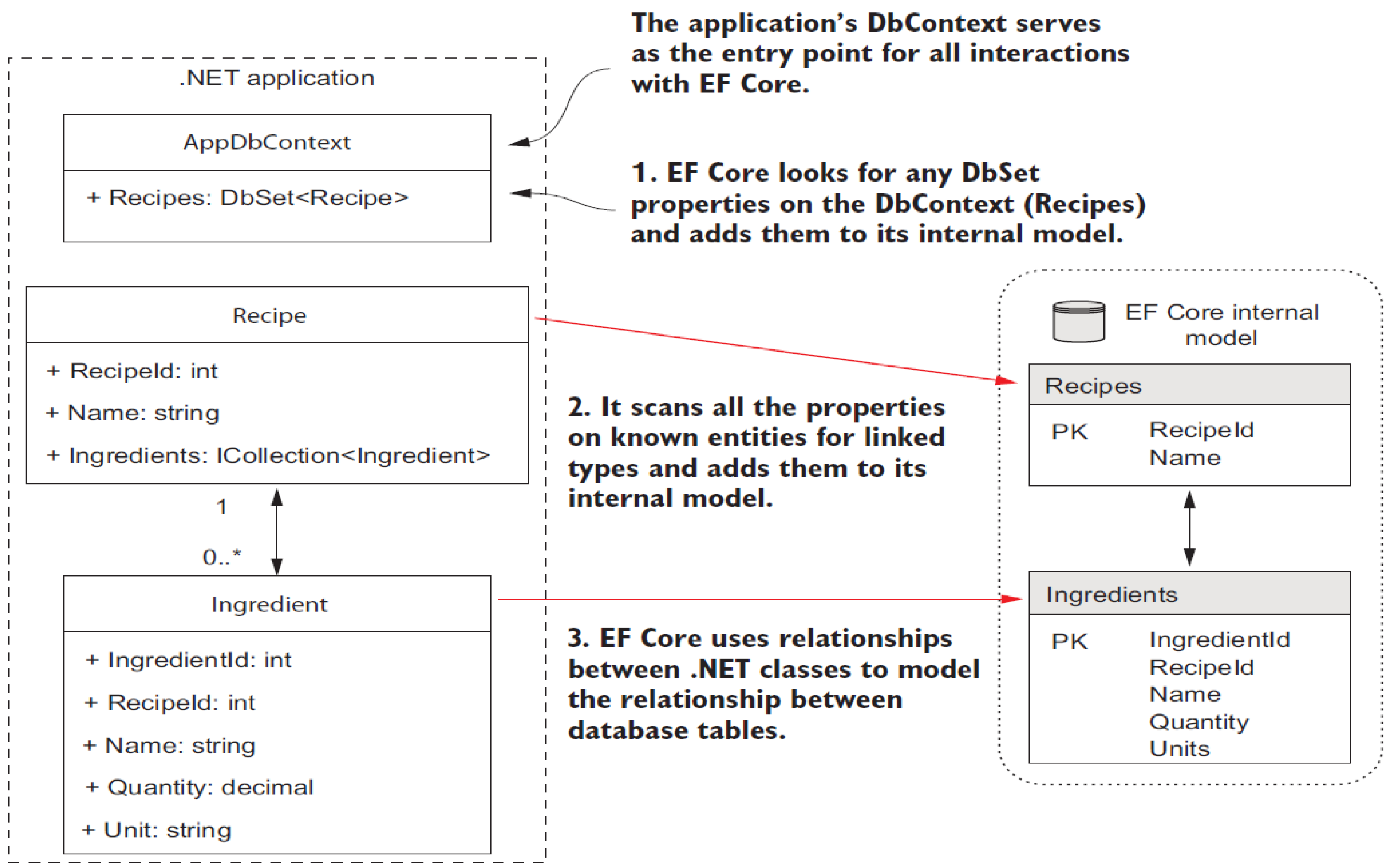
Database Providers

- The Microsoft SQL Provider
  - A very commonly used database provider in EF Core is the **SQL Server** provider. The SQL Server provider enables you to connect a SQL Server database to your application by using EF Core.
- The SQLite Provider
  - **SQLite** is a popular, open source database that is widely used by developers. You can use the SQLite provider to connect an SQLite database to your application by using EF Core.
- Other Providers
  - There are plenty of other database providers. Using these providers, it is possible to connect to various database engines such as **MySQL, Maria DB** and **Db2**.
  - However, notice that while some database providers are maintained by the EF Core Project (Microsoft), others database providers might be maintained by other vendors.

EF Core Project Architecture (1)


Application
Request sent to database provider with application language
Response from database provider with application language
Database Provider
Request sent to database with the database language
Response from the database with database language
Database

EF Core Project Architecture (2)


.NET application
AppDbContext
+ Recipes: DbSet<Recipe>
Recipe
+ RecipeId: int
+ Name: string
+ Ingredients: ICollection<Ingredient>
1
0..*
Ingredient
+ IngredientId: int
+ RecipeId: int
+ Name: string
+ Quantity: decimal
+ Unit: string
The application's DbContext serves as the entry point for all interactions with EF Core.
1. EF Core looks for any DbSet properties on the DbContext (Recipes) and adds them to its internal model.
2. It scans all the properties on known entities for linked types and adds them to its internal model.
3. EF Core uses relationships between .NET classes to model the relationship between database tables.
EF Core internal model
Recipes
PK RecipeId
Name
Ingredients
PK IngredientId
RecipeId
Name
Quantity
Units

Entity Framework Context (1)

- You can use an Entity Framework context to connect to different databases.
- The database to which the Entity Framework context is mapped is determined by a connection string.
- While you can store the connection string as a hard-coded string in your code, a better approach is to store the connection string in a configuration file.
- When working with Entity Framework context and the entities it contains, you might need to update them from time to time.
- To ensure that the context and entities are in sync with the tables and columns in the database, you can use Migrations.

Entity Framework Context (2)

- Proxy between the ASP.NET Core App and the database
- Located in the Data folder (e.g. Data\MVCMovieContext.cs)

```
using Microsoft.EntityFrameworkCore;

namespace MVCMovie.Models
{
    public class MVCMovieContext : DbContext
    {
        public MVCMovieContext(DbContextOptions<MVCMovieContext> options)
            : base(options)
        { }

        public DbSet<Movie> Movie { get; set; }
        public DbSet<Actor> Actor { get; set; }
        public DbSet<ActorMovie> ActorMovie { get; set; }
        public DbSet<Director> Director { get; set; }
    }
}
```

← **Name of the context class**

← **Db sets for the project entities**

Configuration of the DB service

- The DB service and provider is configured in Startup.cs, and linked to the defined Entity Framework context (proxy)

```
public void ConfigureServices(IServiceCollection services)
{
     ...
     services.AddDbContext<MVCMovieContext>(options =>
          options.UseSqlServer(Configuration.GetConnectionString("MVCMovieContext")));
}
```

- Usually, the database connection string is defined in appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MVCMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MVCMovieContext-dc4ea5e6-f835-4702-a883 b6b22d17b019;
                        Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

Related data and navigation properties

- In EF Core you can link an entity to other entities using by navigation properties.
- When an entity is related to another entity, you should add a navigation property to represent the association:
  - When the association is **one to zero-or-one**, the navigation property is represented by a **reference object**. You should also specify a **foreign key**.
  - When the multiplicity of the association is **many** (one-to-many), the navigation property is represented by a **collection**.

Example Movie.cs

- The following code example shows an entity named Movie
  - **One to zero-or-one** relationship with Director, represented by **reference object Director**. Foreign key for the relationship is **DirectorId**;
  - **One-to-many** relationship with ActorMovie represented by the **collection Actors**.

```
public class Movie
{
        public int Id { get; set; }
        public string Title { get; set; }
        public DateTime? ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal? Price { get; set; }
        public decimal? Rating { get; set; }

        public int? DirectorId { get; set; }
        public Director Director { get; set; }

        public ICollection<ActorMovie> Actors { get; set; }
}
```

Example Director.cs

- One-to-many relationship with Movies represented by the collection Movies.

```
public class Director
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }

    [NotMapped]
    public int Age {
        get {
            TimeSpan span = DateTime.Now - BirthDate;
            double years = (double)span.TotalDays / 365.2425;
            return (int)years;
        }
    }
    public string FullName {
        get { return String.Format("{0} {1}", FirstName, LastName); }
    }

    public ICollection<Movie> Movies { get; set; }
}
```

Example Actor.cs

- One-to-many relationship with ActorMovie represented by the collection Movies.

```
public class Actor
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }

    [NotMapped]
    public int Age {
        get {
            TimeSpan span = DateTime.Now - BirthDate;
            double years = (double) span.TotalDays / 365.2425;
            return (int)years;
        }
    }
    public string FullName {
        get { return String.Format("{0} {1}", FirstName, LastName); }
    }

    public ICollection<ActorMovie> Movies { get; set; }
}
```

Example ActorMovie.cs

- Example associative model (junction table) enabling many-to-many relationship between Movie and Actor:
  - **One movie** can be linked to **many actors**;
  - **One actor** can be linked to **many movies**.
- The class has one-to-one relationship with Actor, represented by reference object Actor. Foreign key for the relationship is ActorId.
- One-to-one relationship with Movie, represented by reference object Movie. Foreign key for the relationship is MovieId.

```
public class ActorMovie
{
    public int Id { get; set; }
    public int ActorId { get; set; }
    public Actor Actor { get; set; }
    public int MovieId { get; set; }
    public Movie Movie { get; set; }
}
```

Defining relationships and linking foreign keys

- The relationships need to be specified in the DB context class, so that EF core can link the appropriate tables in the database.

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

        public DbSet<Movie> Movie { get; set; }
        public DbSet<Actor> Actor { get; set; }
        public DbSet<ActorMovie> ActorMovie { get; set; }
        public DbSet<Director> Director { get; set; }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            builder.Entity<ActorMovie>()
                .HasOne<Actor>(p => p.Actor)
                .WithMany(p => p.Movies)
                .HasForeignKey(p => p.ActorId);
                //.HasPrincipalKey(p => p.Id);

            builder.Entity<ActorMovie>()
                .HasOne<Movie>(p => p.Movie)
                .WithMany(p => p.Actors)
                .HasForeignKey(p => p.MovieId);
                //.HasPrincipalKey(p => p.Id);

            builder.Entity<Movie>()
                .HasOne<Director>(p => p.Director)
                .WithMany(p => p.Movies)
                .HasForeignKey(p => p.DirectorId);
                //.HasPrincipalKey(p => p.Id);
        }
    }
```

Using LINQ to Entities

- Language Integrated Query (LINQ) is a set of extension methods that enable you to write complex query expressions.
- You can use these expressions to extract data from databases, enumerable objects, XML documents, and other data sources.
- LINQ to Entities is the version of LINQ that works with EF Core.
- LINQ to Entities enables you to write complex and sophisticated queries to locate specific data, join data from multiple objects, and take other actions on objects from an Entity Framework context.
- You can write LINQ queries in **query syntax**, which resembles SQL syntax, or **method syntax**, in which operations such as “select” are called as methods on objects.

Loading Related Data

- In EF Core you can load related entities by using navigation properties. To load related data, you need to choose an ORM pattern.
- EF Core contains several ORM patterns, which include:
  - **Explicit loading**. Using this pattern, the related data is loaded explicitly from the database after the original query is completed;
  - **Eager loading**. Using this pattern, the related data is loaded from the database as part of the original query.
  - **Lazy loading**. Using this pattern, the related data is loaded from the database as you access the navigation property in the C# code.

Explicit Loading (LINQ syntax)

- To load related entities by using the explicit loading ORM pattern, you should use the **Entry** method of the Entity Framework context class.

```
Movie movie = _context.Movie
     .Single(c => c.Id == 1);
```

Load the Movie entry with Id == 1

```
_context.Entry(movie)
    .Collection(c => c.Actors)
    .Load();
```

Then load the Collection Actors from the movie entry

```
_context.Entry(movie)
    .Reference(c => c.Director)
    .Load();
```

And then load the Reference Director from the movie entry

Eager Loading (LINQ syntax)

- To load related entities by using the eager loading ORM pattern you need to use the **Include** method. The **Include** method specifies related entities to be included in the query results.
- If you need to include more levels of related data, you can use the **ThenInclude** method. The **ThenInclude** method can be used to drill down through the relationships.

```
Movie movie = await _context.Movie
               .Include(m => m.Director)
               .Include(m => m.Actors).ThenInclude(m => m.Actor)
               .FirstOrDefaultAsync(m => m.Id == id);
```

Manipulating Data by Using Entity Framework

- EF Core can track entities that you retrieve from the database.
- EF Core uses change tracking so that when you call the SaveChanges method on the Entity Framework context object, it can synchronize your updates with the database.
- Each entity in EF Core can be in one of the following states:
  - **Added**. The entity was added to the context and does not exist in the database;
  - **Modified**. The entity was changed since it was retrieved from the database.
  - **Unchanged**. The entity was not changed since it was retrieved from the database.
  - **Deleted**. The entity was deleted since it was retrieved from the database.

Inserting New Entities

- To add a new entity in EF Core:
    1. Add the entity to the DB context
    2. Update/reflect the changes in the database

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("Id,Title,ReleaseDate,Genre,Price,Rating,DirectorId")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    ViewData["DirectorId"] = new SelectList(_context.Director, "Id", "FullName", movie.DirectorId);
    return View(movie);
}
```

Deleting an Entity

- To delete an existing entity in EF Core:
  1. Delete the entity from the DB context
  2. Update/reflect the changes in the database

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Updating an Entity

- To update an existing entity in EF Core:
  1. Update the entity in the DB context
  2. Update/reflect the changes in the database

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("Id,FirstName,LastName,BirthDate")] Director director)
{
    if (id != director.Id) { return NotFound(); }
    if (ModelState.IsValid) {
        try {
            _context.Update(director);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException) {
            if (!DirectorExists(director.Id)) { return NotFound(); }
            else { throw; } }
        return RedirectToAction(nameof(Index));
    }
    return View(director);
}
```

Using Migrations

- Working with an application that interacts with a database requires you to create models and sometimes, you might find yourself changing or upgrading those models according to the application needs.
  - EF Core **will not** update the database automatically.
- Using migrations, you will be able to create a database, upgrade it and manipulate it according to your application models.
- Entity Framework Core Package Manager Console Tools
  - **Add-Migration Name**
    - Ads a new migration with name Name
    - Creates a snapshot of the Entity model and database
  - **Update-Database (-Migration Name)**
    - Creates the schema of the Database
    - Updates the database with the new/updated schema

```
context.Director.AddRange(
    new Director { /*Id = 1, */FirstName = "Rob", LastName = "Reiner", BirthDate = DateTime.Parse("1947-3-6") },
    new Director { /*Id = 2, */FirstName = "Ivan", LastName = "Reitman", BirthDate = DateTime.Parse("1946-11-27") }
);
context.SaveChanges();

context.Actor.AddRange(
    new Actor { /*Id = 1, */FirstName = "Billy", LastName = "Crystal", BirthDate = DateTime.Parse("1948-3-14") },
    new Actor { /*Id = 2, */FirstName = "Meg", LastName = "Ryan", BirthDate = DateTime.Parse("1961-11-19") },
    new Actor { /*Id = 3, */FirstName = "Carrie", LastName = "Fisher", BirthDate = DateTime.Parse("1956-10-21") }
);
context.SaveChanges();

context.Movie.AddRange(
    new Movie {
        //Id = 1,
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Rating = 5,
        Price = 7.99M,
        DirectorId = context.Director.Single(d => d.FirstName == "Rob" && d.LastName == "Reiner").Id
    },
    new Movie {
        //Id = 2,
        Title = "Ghostbusters",
        ReleaseDate = DateTime.Parse("1984-3-13"),
        Genre = "Comedy",
        Rating = 6,
        Price = 8.99M,
        DirectorId = context.Director.Single(d => d.FirstName == "Ivan" && d.LastName == "Reitman").Id
    } );
context.SaveChanges();

context.ActorMovie.AddRange(
    new ActorMovie { ActorId = 1, MovieId = 1 },
    new ActorMovie { ActorId = 2, MovieId = 1 },
    new ActorMovie { ActorId = 3, MovieId = 1 },
);
context.SaveChanges();
```

Seeding Data

Practical

- MVCMovie2-2023.rar – uploaded to e-kursevi
- For additional training, the tutorial at: https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-6.0