ASP Data Access
Entity Framework Core

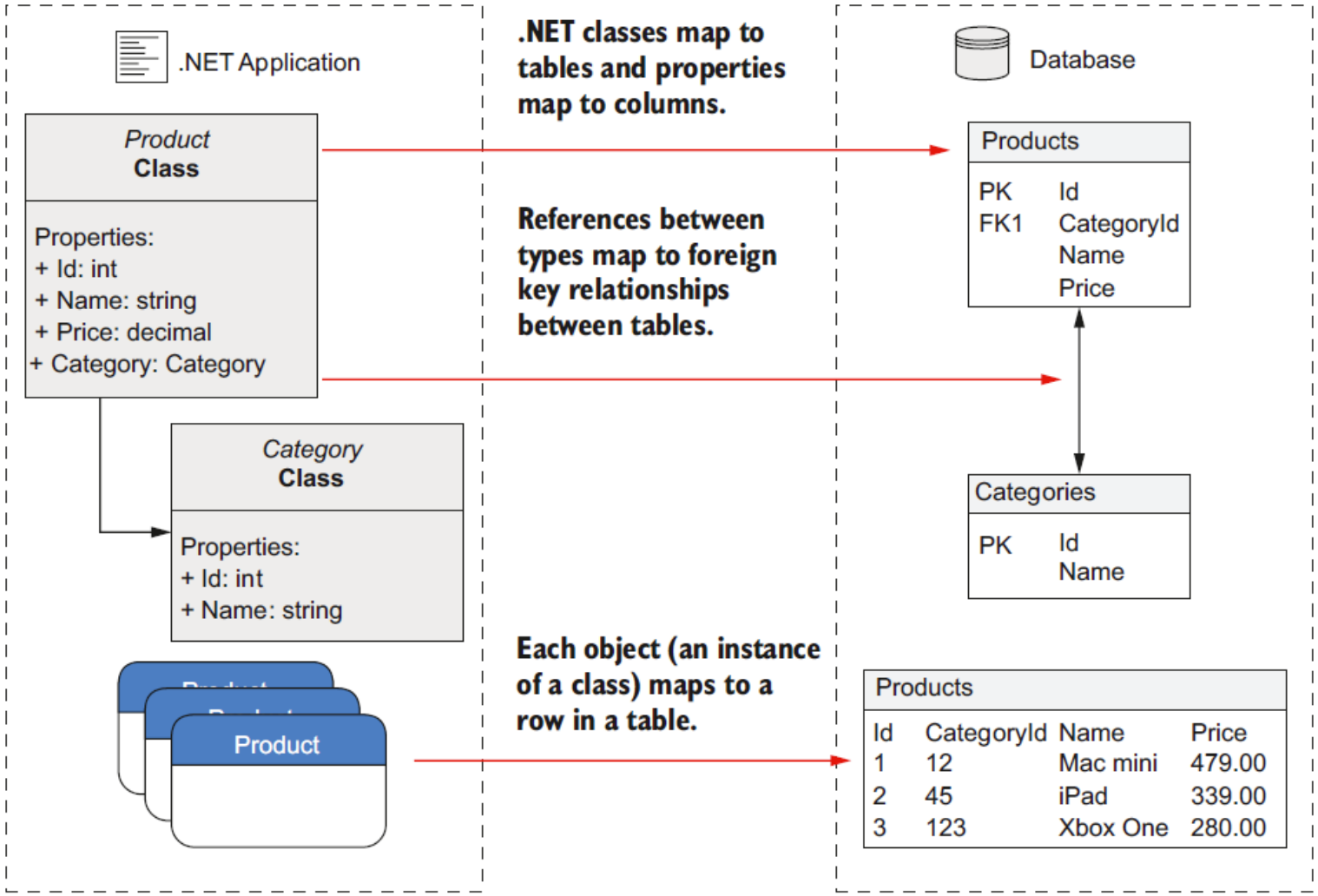
Commonly use relational databases

- A relational database is a digital database based on the relational model of data, as proposed by E. F. Codd in 1970
- A software system used to maintain relational databases is a relational database management system (**RDBMS**).
- Virtually all relational database systems use **SQL** (Structured Query Language) for querying and maintaining the database
- Common RDBMS;
  - Microsoft SQL server
  - Oracle DB
  - IBM DB2
  - MySQL / MariaDB
  - Postgres
  - *Many other exist*

What is EF Core?

- EF Core is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects

- With the Entity Framework, developers can work at a higher level of abstraction when they deal with data, and can create and maintain data-oriented applications with less code than in traditional applications



Which databases?

- EF Core supports many database engines:

Database	NuGet package to install
Microsoft SQL Server	Microsoft.EntityFrameworkCore.SqlServer
SQLite	Microsoft.EntityFrameworkCore.SQLite
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL
IBM Data Server (DB2)	IBM.EntityFrameworkCore (on Linux, install the IBM.EntityFrameworkCore-lnx NuGet package)
MySQL	MySql.Data.EntityFrameworkCore –Pre Or Pomelo.EntityFrameworkCore.MySql
SQL Server Compact Edition 4.0	EntityFrameworkCore.SqlServerCompact40
EF Core In-Memory Database (usefull for testing)	Microsoft.EntityFrameworkCore.InMemory
Firebird database	EntityFrameworkCore.FirebirdSQL
Oracle	Not yet / paid version exist

Code first vs. Database first

- Code first
  - For new projects where no database exists
  - The developer defines the model (entities) in C# and EF generates a matching database schema

- Database first
  - Used when the database schema already exist or if you prefer to design the database yourself with a tool or SQL statements (DDL)

EF DB FIRST

Database first How-to

1. Open SQL Server Object Explorer
2. Under SQL Server select (localdb)\MSSQLLocalDB …

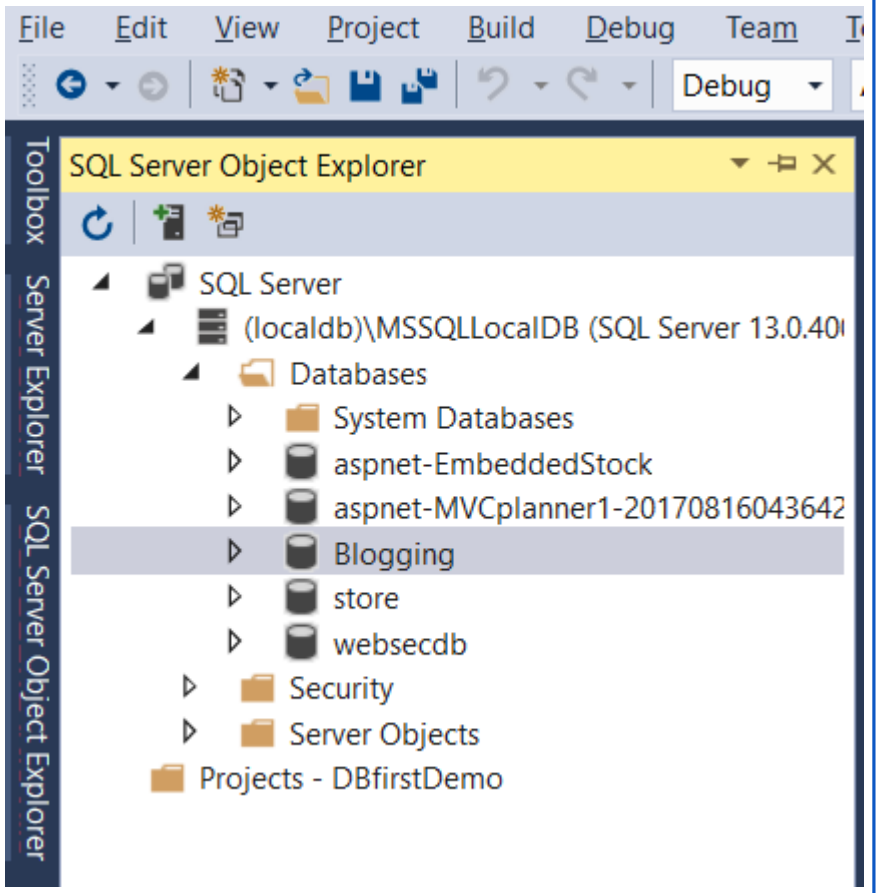
If you want to create a new Database and/or Schema:

Right-click on the database in **Server Explorer**

and select **New Query**

Enter a SQL script into the query editor (as shown next page)

And right-click on the query editor and select **Execute**



SQL for defining a Schema

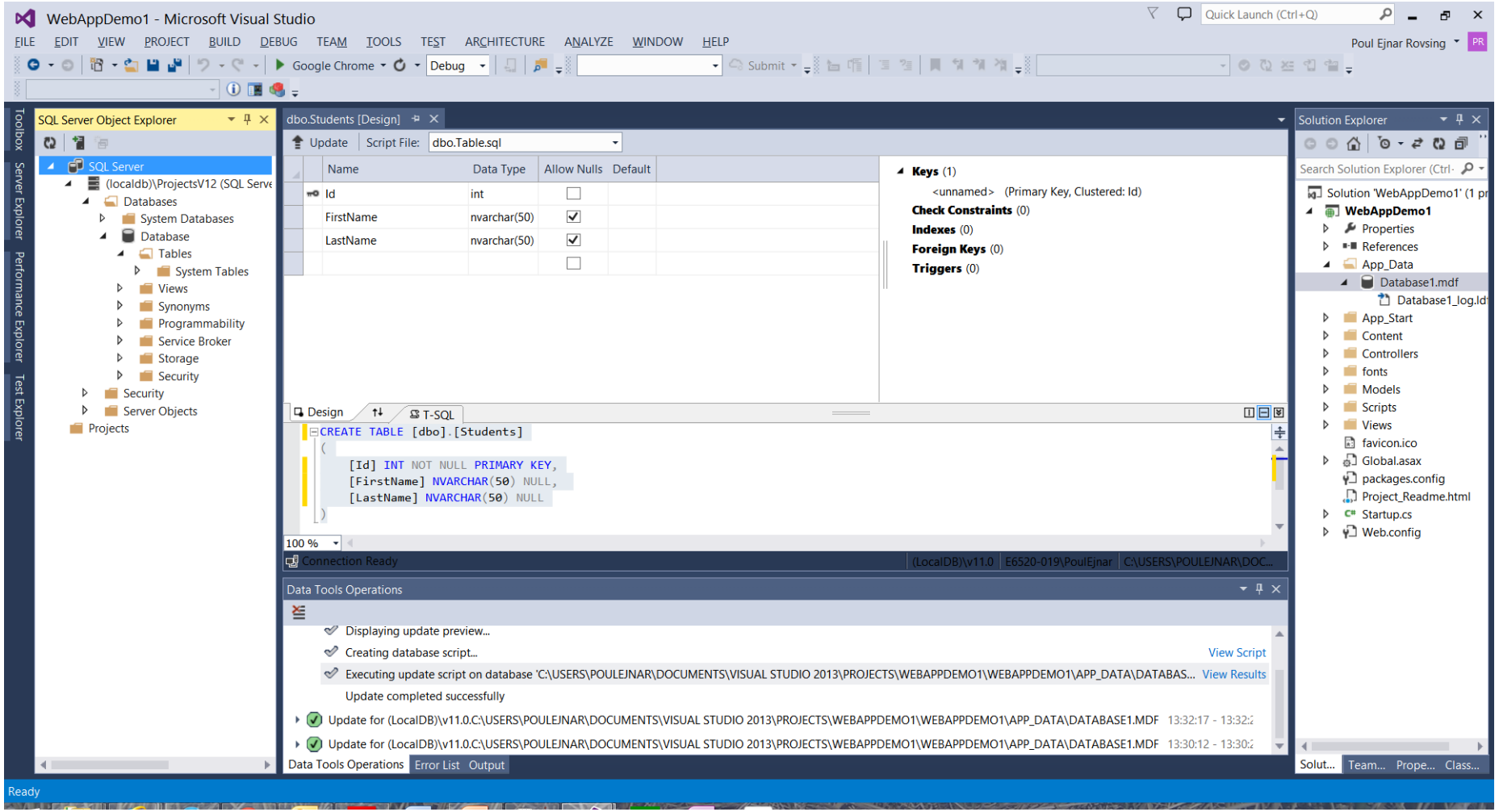
```
CREATE DATABASE [Blogging];
GO

USE [Blogging];
GO

CREATE TABLE [Blog] (
    [BlogId] int NOT NULL IDENTITY,
    [Url] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_Blog] PRIMARY KEY ([BlogId])
);
GO

CREATE TABLE [Post] (
    [PostId] int NOT NULL IDENTITY,
    [BlogId] int NOT NULL,
    [Content] nvarchar(max),
    [Title] nvarchar(max),
    CONSTRAINT [PK_Post] PRIMARY KEY ([PostId]),
    CONSTRAINT [FK_Post_Blog_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blog] ([BlogId]) ON DELETE CASCADE
);
GO
```

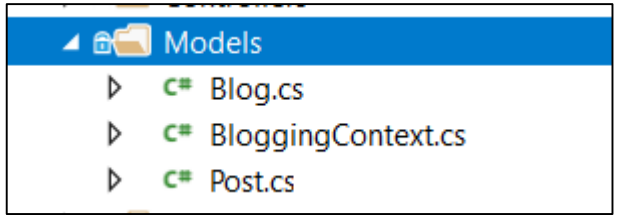
OR Design your Database with a design tool



Reverse engineer your model

- In Tools –> NuGet Package Manager –> Package Manager Console run the following command to create a model from an existing database:

```
Scaffold-DbContext
"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```




```
CREATE TABLE [Blog] (
   [BlogId] int NOT NULL IDENTITY,
   [Url] nvarchar(max) NOT NULL,
   CONSTRAINT [PK_Blog] PRIMARY KEY ([BlogId])
);
```

```
public partial class Blog
{
   public Blog()
   {
      Post = new HashSet<Post>();
   }

   public int BlogId { get; set; }
   public string Url { get; set; }
   public ICollection<Post> Post { get; set; }
}
```

If you don't use Visual Studio:

```
dotnet ef dbcontext scaffold [ConnectionString] [Provider] [options]
```

DB context

- The context represents a session with the database and allows you to query and save instances of the entity classes

```
public partial class BloggingContext : DbContext
{
  public virtual DbSet<Blog> Blog { get; set; }
  public virtual DbSet<Post> Post { get; set; }

 protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
 {
   if (!optionsBuilder.IsConfigured)
   {
      // Your connection string, you should move it out of source code!
      optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;");
   }
}
```

EF CODE FIRST

EF with Code First – Add a Model

- Right click the Models folder and select Add class

```
using System;
using System.Data.Entity;

namespace SportsStore.Models
{
    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

The database context class

- The DbContext base class provides access to the Entity Framework Core's underlying functionality, and the Products property will provide access to the Product objects in the database

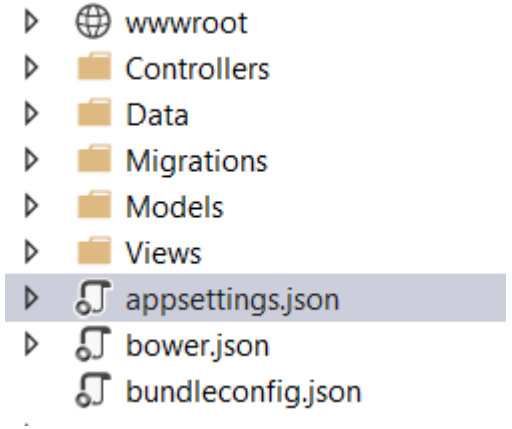
```
public class ApplicationDbContext : DbContext {

    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options) { }

    public DbSet<Product> Products { get; set; }
}
```

Create a Connection String

- Open the `appsettings.json` file



- Add the following connection string to settings file:

```
"ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=SportsStore;Trusted_Connection=True;MultipleActiveR
esultSets=true"
  },
```

Configuring the Application

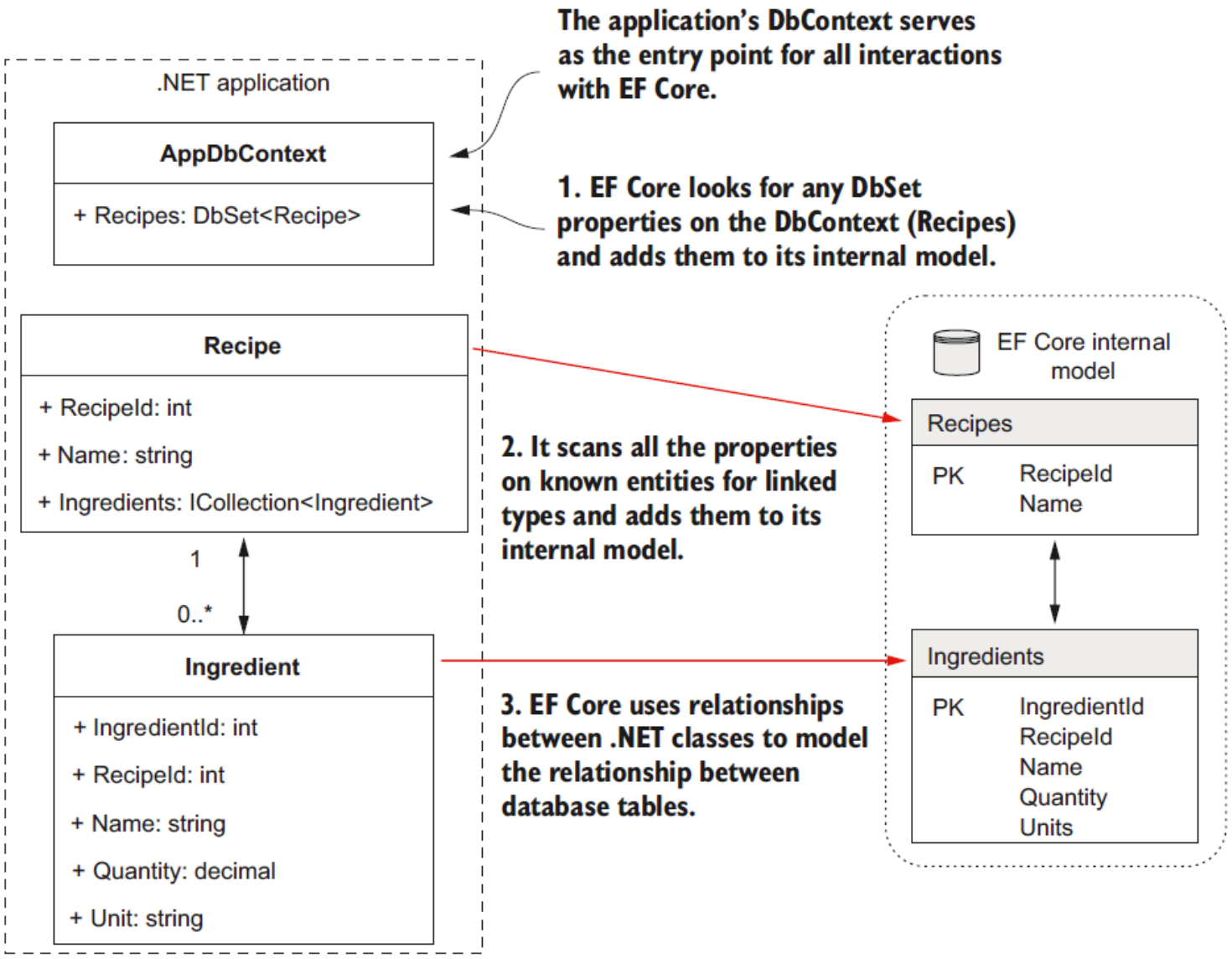
- Read the connection string and configure the application to use it to connect to the database

```
public class Startup {
    public Startup(IConfiguration configuration) {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
        services.AddDbContext<ApplicationDbContext >(options =>
            options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
        services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
        . . .
        SeedData.EnsurePopulated(app);  // Only if seeding data manually
    }
}
```

EF Core's internal model



Database Migration

- Entity Framework Core is able to generate the schema for the database using the model classes through a feature called ***migrations***
- When you prepare a migration, EF Core creates a C# class that contains the SQL commands required to prepare the database
- If you need to modify your model classes, then you can create a new migration that contains the SQL commands required to reflect the changes
- EF Core commands are performed using the Package Manager Console (or Powershell/Bash/?)

PMC:

```
Add-Migration InitialSchema
Update-Database
```

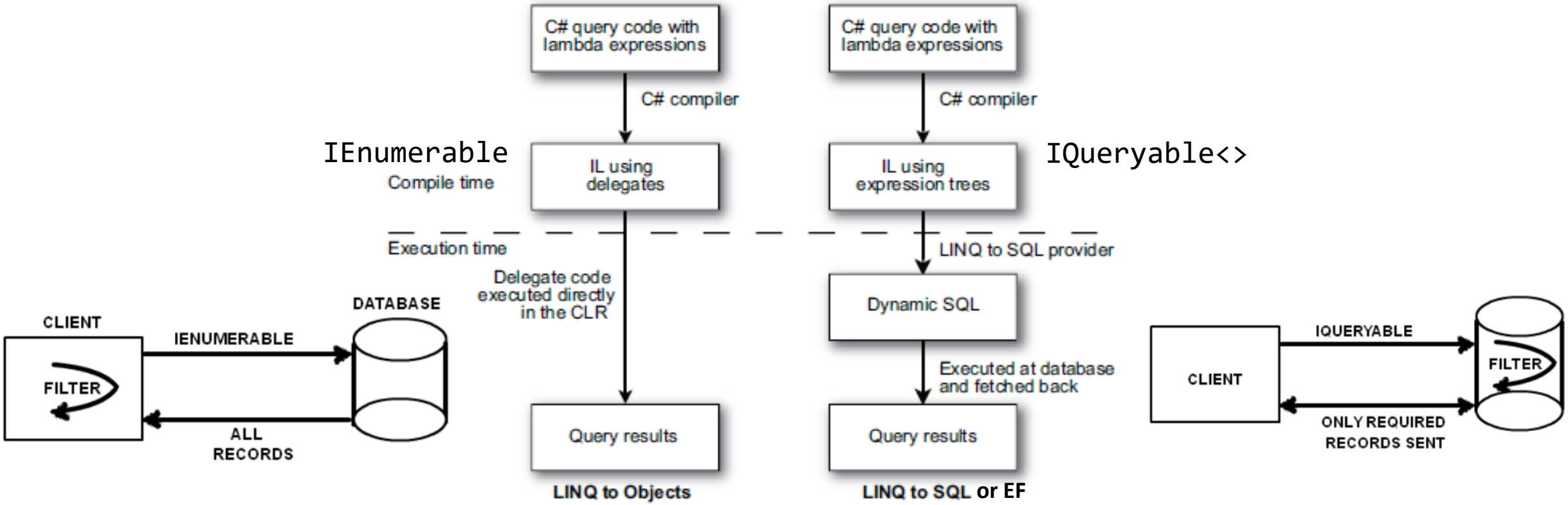
Shell:

```
dotnet ef migrations add InitialSchema
dotnet ef database update
```

- When this command has finished, you will see a Migrations folder in the Visual Studio Solution

The Use of Expression Trees

- Both LINQ to Objects and LINQ to EF start with C# code and end with query results
- The ability to execute the code remotely as LINQ to EF does comes through expression trees



Paging

```
public ViewResult List(int page = 1) {
    return View(new ProductsListViewModel {
        Products = repository.Products
            .OrderBy(p => p.ProductID)
            .Skip((page - 1) * PageSize)
            .Take(PageSize),
        PagingInfo = new PagingInfo {
            CurrentPage = page,
            ItemsPerPage = PageSize,
            TotalItems = repository.Products.Count()
     }
});
```

```
public interface IProductRepository {
    IEnumerable<Product> Products { get; }
}
```

Testing with InMemory

- The InMemory provider is useful when you want to test components using something that approximates connecting to the real database, without the overhead of actual database operations
- InMemory is designed to be a general purpose database for testing, and is not designed to mimic a relational database

Testing with InMemory

```
public void Add_writes_to_database()  {
    var options = new DbContextOptionsBuilder<BloggingContext>()
            .UseInMemoryDatabase(databaseName: "Add_writes_to_database")
            .Options;

    // Run the test against one instance of the context
    using (var context = new BloggingContext(options)) {
        var service = new BlogService(context);
        service.Add("http://sample.com");
    }

    // Use a separate instance of the context to verify correct
    // data was saved to database
    using (var context = new BloggingContext(options)) {
                Assert.AreEqual(1, context.Blogs.Count());
                Assert.AreEqual("http://sample.com", context.Blogs.Single().Url);
            }
        }
```

Post redirect get pattern

Without POST REDIRECT GET PATTERN

1.GET “Products/Create”
2.User types in some information
3.POST “Products/Create”
4.Validation fails, re-display the form with warnings – user corrects the input
5.POST “Products/Create”
6.Validation passes, item is saved and message appears “Saved Okay”
7.User refreshes the page, this results in **POST** “Products/Create”
8.Validation passes again, ***item is saved again*** and message appears again. Oops!

With POST REDIRECT GET PATTERN

1.GET “Products/Create”
2.User types in some information
3.POST “Products/Create”
4.Validation fails, re-display the form with warnings – user corrects the input
5.POST “Products/Create”
6.Validation passes, item is saved redirect, using a **GET** to “Products/View/5”
7.User refreshes the page, this results in a harmless **GET** “Products/View/5”

https://www.stevefenton.co.uk/2011/04/asp-net-mvc-post-redirect-get-pattern/

Data Seeding

```
protected override void OnModelCreating(ModelBuilder builder)
{
  base.OnModelCreating(builder);
  // Add your customizations
  builder.ApplyConfiguration(new RegistrationConfig());

  // Seed data
  builder.Entity<Blog>().HasData(
    new Blog { BlogId = 1, Url = "http://sample.com" });
  builder.Entity<Post>().HasData(
    new { BlogId = 1, PostId = 1, Title = "First post", Content = "Test 1" },
    new { BlogId = 1, PostId = 2, Title = "SecondPost", Content = "Test 2" });
}
```

- To add entities that have a relationship the foreign key values need to be specified. Frequently the foreign key properties are in shadow state, so to be able to set the values an anonymous class should be used.
- Alternatively, you can use context.Database.EnsureCreated() to create a new database containing the seed data, for example for a test database or when using the in-memory provider.

Seed the database - alternative

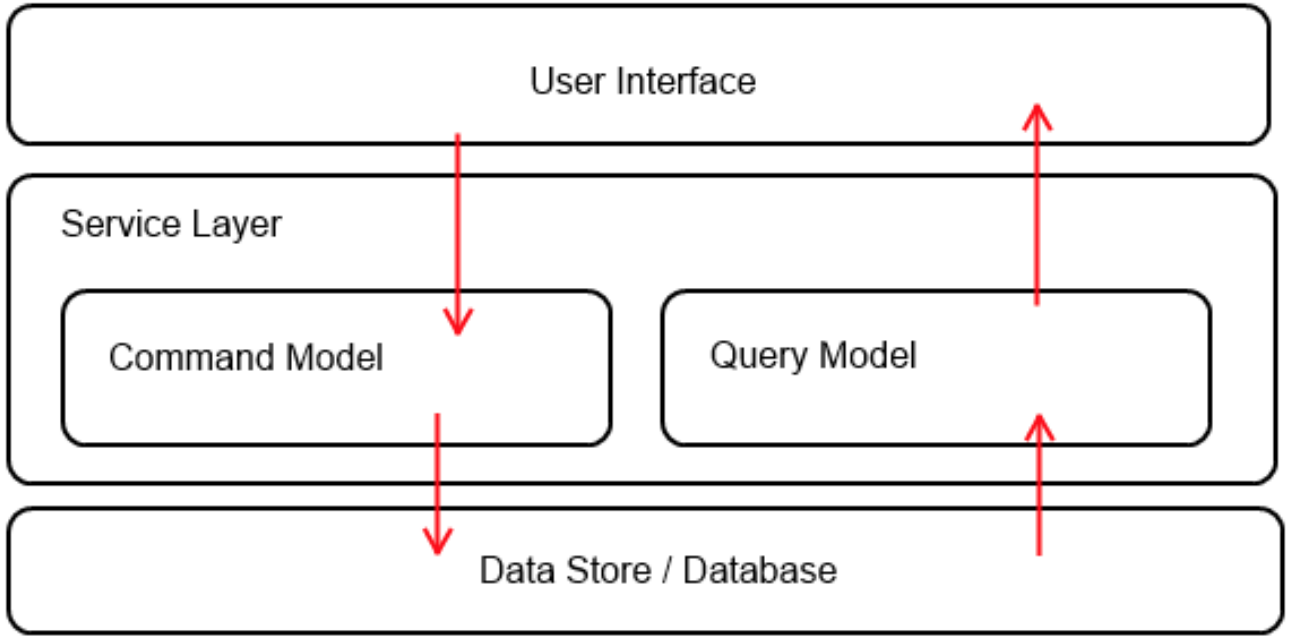
- Add a class to populate the database with some sample data

```
public static class SeedData {

    public static void EnsurePopulated(IApplicationBuilder app) {
            ApplicationDbContext context = app.ApplicationServices
                .GetRequiredService<ApplicationDbContext>();
            if (!context.Products.Any()) {
                context.Products.AddRange(
                    new Product {
                        Name = "Kayak", Description = "A boat for one person",
                        Category = "Watersports", Price = 275
                    },
                . . .
    }
}
```

COMMAND QUERY SEPARATION

- Queries: Return a result and do not change the observable state of the system (are free of side effects)
- Commands: Change the state of a system but do not return a value
- Because the term 'command' is widely used in other contexts refer to them as 'modifiers', you may also see the term 'mutators'



http://martinfowler.com/bliki/CommandQuerySeparation.html

Object-object maping

OWN MAPPER CODE:

```
using ditmer.ProjektWeb3.Domain.Database;
using ditmer.ProjektWeb3.Domain.Models.Api;

namespace ditmer.ProjektWeb3.Business.Extensions
{
    public static partial class MapExtension
    {
        public static Task ToTask(this TaskModel model)
        {
            if (model == null)
            {
                return null;
            }

            var task = new Task
            {
                Description = model.Description,
                Title = model.Title,
                Inactive = model.Inactive,
                EstimatedHours = model.EstimatedHours,
                RemainingHours = model.RemainingHours,
                Completed = model.Completed
            };

            return task;
        }

    }
}
```

AutoMapper is an object-object mapper.
Object-object mapping works by transforming an input object of one type into an output object of a different type.

```
AutoMapper.Mapper.CreateMap<Book, BookViewModel>();
var model = AutoMapper.Mapper.Map<BookViewModel>(book);

AutoMapper.Mapper.CreateMap<Book, BookViewModel>()
          .ForMember(dest => dest.Author,
         opts => opts.MapFrom(src => src.Author.Name));
```

References & Links

- **ASP.Net Core in Action** chapter 12
- Entity Framework Core in Action
- **Getting Started with EF Core on ASP.NET Core with a New database**
  https://docs.microsoft.com/da-dk/ef/core/get-started/aspnetcore/new-db
- Documentation
  https://docs.microsoft.com/en-us/ef/core/index
- DbUp is a .NET library that helps you to deploy changes to SQL Server databases
  https://dbup.readthedocs.io/en/latest/
- Scaffolding Entity Framework Core with CatFactory
  https://www.codeproject.com/Articles/1160615/Scaffolding-Entity-Framework-Core-with-CatFactory

- Using MongoDB .NET Driver with .NET Core WebAPI
  https://www.codeproject.com/Articles/1151842/Using-MongoDB-NET-Driver-with-NET-Core-WebAPI