Backend designing with Entity Framework Core

Database access for web application

Veikko Lintujärvi

BACHELOR'S THESIS
December 2019

Information and communication technology
Software engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Information and communication technology
Software engineering

VEIKKO LINTUJÄRVI
Backend designing with Entity Framework Core
Database access for web application

Bachelor's thesis 29 pages, appendices 4 pages
December 2019

This bachelor’s thesis reviews Entity Framework Core when used as an Object-Relational Mapper in a backend application. Study will go into detail describing the capabilities of the framework and raises discussion over the features, usability and maturity of it.

Preview and discussion is connected by examples to a project called Shop Management Tool. Shop Management Tool is a web application used in a general store and café for monitoring and recording money flow and warehouse balances as well as presenting data in understandable format.

Technical capabilities of the framework is compared to the requirements of the problem at hand. The thesis will go through how different features were used to carry out the needed tasks and what impact the possible flaws in them caused for the development of the end product itself.

By the course of the thesis Entity Framework Core proved to be useful part of the backend application. Even though some of the minor functionalities weren’t fully complete, it contained sufficient set of tools needed to perform the task of designing data structure and accessing to database. Overall the user experience was pleasant.

Key words: ORM, Entity Framework Core, .NET Core, ASP.NET Core, backend

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Tieto- ja viestintätekniikka
Ohjelmistokehitys

VEIKKO LINTUJÄRVI
Backend designing with Entity Framework Core
Database access for web application

Opinnäytetyö 29 sivua, liitteet 4 sivua
Joulukuu 2019

Tämä opinnäytetyö käsittelee Entity Framework Corea käytettynä Object-Relational Mapperinä backend-sovelluksessa. Tutkielma esittelee yksityiskohtaisesti Entity Framework Coren kyvykkyyksiä ja pohtii sen ominaisuuksia, käytettävyyttä ja kypsyyttä projektina.

Esittely ja pohdinta yhdistyvät esimerkein projektiin nimeltä Shop Management Tool. Shop Management Tool on verkossa julkaistava sovellus, jota käytetään sekä päivittäistavarakaupan ja kahvilan rahavirran ja varastosaldojen monitorointiin ja tallentamiseen että kerätyn datan esittämiseen asiakkaalle ymmärrettävässä muodossa.

Tutkielman mittaan Entity Framework Coren teknisiä kyvykkyyksiä verrataan tarpeisiin, joihin sen tulisi vastata. Opinnäytetyössä käydään lävitse, miten eri ominaisuuksia käytettiin vaadittavissa tehtävissä ja mikä vaikutus niissä mahdollisesti esiintyvillä puutteilla oli itse lopputuotteen kehittämisprosessiin.

Opinnäytetyön mittaan Entity Framework Core osoittautui hyödylliseksi osaksi backend-sovellusta. Vaikkakin osa vähäisemistä toiminnallisuuksista olikin vielä kesken, piti se kuitenkin sisällään riittävän valikoiman työkaluja, jotta voitiin tuottaa ohjelman tietorakenne sekä yhteys tietokantaan. Kaiken kaikkiaan kokemus Entity Framework Coren käytöstä oli positivinen.

Asiasanat: ORM, Entity Framework Core, .NET Core, ASP.NET Core, backend

CONTENTS

GLOSSARY

ORM	Object relational mapper.
EF Core	Entity Framework Core.
IDE	Integrated Development Environment.
Code Snippet	Template of code applied by command in IDE.
Business logic	Part of program that encodes the real-world business rules that determine how data can be created, stored, and changed.
Orthogonal	Adjective of modularity. When orthogonal part changes, no other part is affected.
Database engine	Underlying software component that a database management system uses to create, read, update and delete data from a database.
Mocking	To simulate parts of program in order to create functional test environment.
SQL	Structured Query Language. Widely used querying databases.
de facto	Widely used or expected choice of method or technology to a given task.
CRUD	Create, Read, Update, Delete. Basic functions of persistent storage.
POCO	Plain old CLR object.
HTTP	Hypertext Transfer Protocol.
VS Code	Visual Studio Code.
LINQ	Language Integrated Query
UI	User interface

1 INTRODUCTION

Entity Framework Core is an open source, modern Object-Relational Mapper made to work with ASP.NET Core framework offering a full set of functionalities to interact with database without writing a single line of SQL code. One could say that ORMs have become a de facto way to communicate between backend application and database in recent years and many options for various programming languages and frameworks have been developed with great popularity.

In my thesis I am going to go through the backend design as well as EF Core's nuances as a part of backend application and programming project producing management tool for a general store and café. Thesis is sectioned in three parts. First is about EF Core itself going through the technical capabilities and the main concepts introduced in the framework. In the second one I dive into the backend project including some decisions I made during the course of the development and why. The third one wraps itself around more abstract concepts used not only by EF Core but other modern backend applications as well. Understanding these concepts is as crucial as understanding the technology if you want to create well-formed software.

On the next page I'll go through the reasons why this backend application was created in the first place. In my opinion the most important thing in the programming project is the motivation. When you are working for something you believe in everything feels a little bit lighter.

2 BACKGROUND

Livonsaaren Osuuskauppa is a general store, café and meeting place for locals, cottage owners as well as random passer-bys. It was a collaborative effort to get much needed marketplace for the local people and thus crowd funding proved to be very successful way to finance the affair. The whole product storage is maintained with the shares of the shop cooperative members. Currently (11.8.2019) there is 40 of those.

Even though many people have been very generous by putting their money into the product storage and property, there wasn't much excess capital to put on the cashier system software or such. Since I also wanted to contribute to the common effort, I offered my helping hand and by promising functional digital solution to handle the cash flow and data for accounting purposes. The task itself wasn't impossible but the time schedule made it harder than I thought.

At the beginning all this seemed quite over-whelming but then I got lucky. I was talking about this new project with one of my colleagues and he offered to help. I gladly took him in. So we made this separation of duties: I will be responsible for the backend and he will handle the front-end. Pretty straightforward. This is also the reason why I will be going through backend side of the solution in this thesis.

Now in retrospective I couldn't have made it without him. We had roughly one and a half months of time and we needed to build fully functional web application from the scratch, implement it to the production (meaning not only software but also cashier accessories like bar code scanners and receipt printers) and adding needed product information into the database. And it was just the technical side of it. We also needed to build up some level of project management and customer service structure as well.

But I wouldn't be writing this report if the project didn't succeed. Though succeeding is relative on this era of agile development and never ending DevOps projects. Also this project doesn't seem to be ending any time soon. I pushed my latest commit to GitHub repository just yesterday evening.

3 ENTITY FRAMEWORK CORE

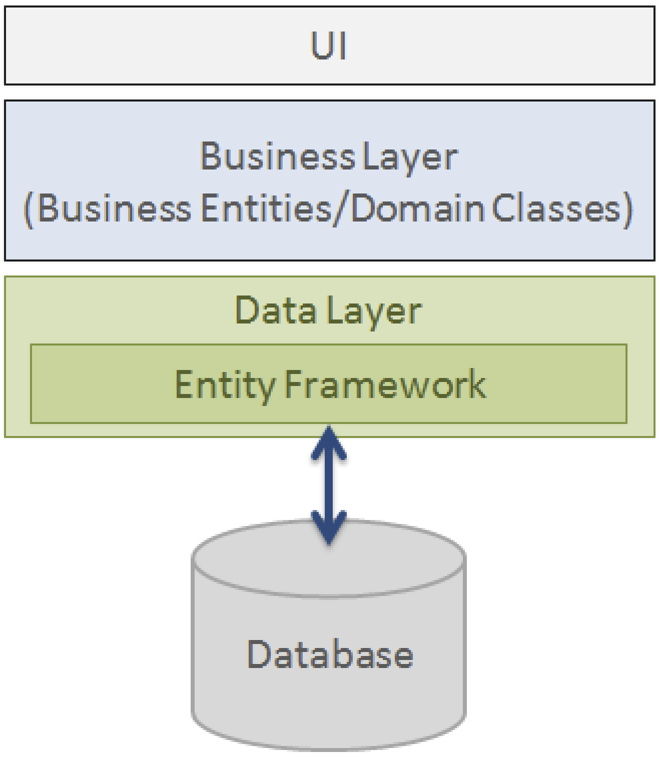
This section is purely dedicated to go through the essential parts of the Entity Framework Core.

3.1. Project

Entity Framework Core is an open source project made to imitate proprietary Entity Framework project. Where Entity Framework was made for Windows and .NET Framework, EF Core is its cross-platform equivalent for .NET Core (Should I use EF6 or EF Core).

3.1.1 Comparison with Entity Framework 6

Even though the goals of the Entity Framework 6 and Entity Framework Core are similar, the latter includes many features specific only to it. On the other hand EF 6's features are not and never will be included to its open source counterpart. Comparing these two feature by feature is out of the scope of this thesis but you can find a high-level list in Appendix 1. for further reading.



GRAPH 3.1 Place of Entity Framework in the web application (What is Entity Framework? Internet article.)

For the most parts the differences between these projects are pretty trivial. Both of them does the trick if you need basic ORM for your .NET project (graph 3.1.). I recommend to favor EF Core if you have a possibility to choose. There is a couple of reason for this. Firstly EF Core is more flexible when it comes to platforms. You can move your EF Core application from Windows based server to a Linux without any additional configurations. Secondly EF Core has now the momentum and new versions are pushed out frequently. Microsoft team even announced that .NET Core is going to be the future .NET Framework (.NET Core is the future of .NET) and EF Core is the ORM-of-choice for .NET Core. Lastly I find EF Core just more intuitive to use.

3.1.2 Aim

EF Core contains the basic functionalities of an ORM. Here is a list and short introduction of those:

- Context as a structured object-orientated representation of the database (3.2.3 DbContext).

- Mapping between objects and rows in database table (3.2.2 Entities).
- Possibility to manipulate database schema, data it contains and add calculation without explicit SQL-queries (3.2.1 CRUD).
- Database schema engineering (3.2.4 Code-first approach) and reverse-engineering (3.2.5 Scaffolding) from or to source code.

In addition to the above EF Core contains set of more advanced techniques like lazy-loading (3.3.2 Loading related data) and migrations (3.3.3). I will go through those and couple of other concepts in the section 3.3. Often these advanced techniques are offered as a stand-alone projects but in EF Core they are included natively.

3.2. Basic concepts

Entity Framework Core has mostly the same concepts included into it as all the other ORMs: POCO classes mapping to the database tables, ready-made functions to communicate with the database and model generation from the database schema (Object Relational Mapping Concepts). In the next chapter I'll go through these concepts.

3.2.1 CRUD

CRUD is an abbreviation of the basic actions to interact with the database: CREATE, READ, UPDATE and DELETE. These are the functions which web applications backend should be able to carry out or else you cannot call it such (What is CRUD).

Let's go through these functionalities shortly on a very general level. Create means for the backend of the application to receive needed information from the frontend via API and with that form and push a new row into database under a correct table or tables. Read means to access any data inside the database in orderly manner. Update overrides old data with the new one and delete removes rows.

In EF Core as in all of the modern ORMs these basic actions are in the center of the functionality design. All of these are done via Context (3.2.3 DbContext). They are just functions under that instance and thus can be only applied to the entities under it.

In many cases the whole purpose of the web application's backend is to offer these abilities via API to the frontend. The other capabilities such as data validation or calculation are added later on if at all.

3.2.2 Entities

Entity is a atomic part of EF Core's model. You can think entity as a one row in the database. Entities are data carriers between the backend application and database.

From the MVC (4.1. Model-Controller-View) point of view entities are located in the model (4.1.1 Model) and are the essential part of it. You could say that the entities in the model are to the backend application what schema is for the database: modular representation of the data structure (The Entity Framework Core Model).

There's three ways to configure these entities: convention, data annotations and Fluent API. In the center of this are Context (3.2.3 DbContext) and POCOs (Code 3.1). Abbreviation POCO derives from plain old CLR object meaning object containing only properties.

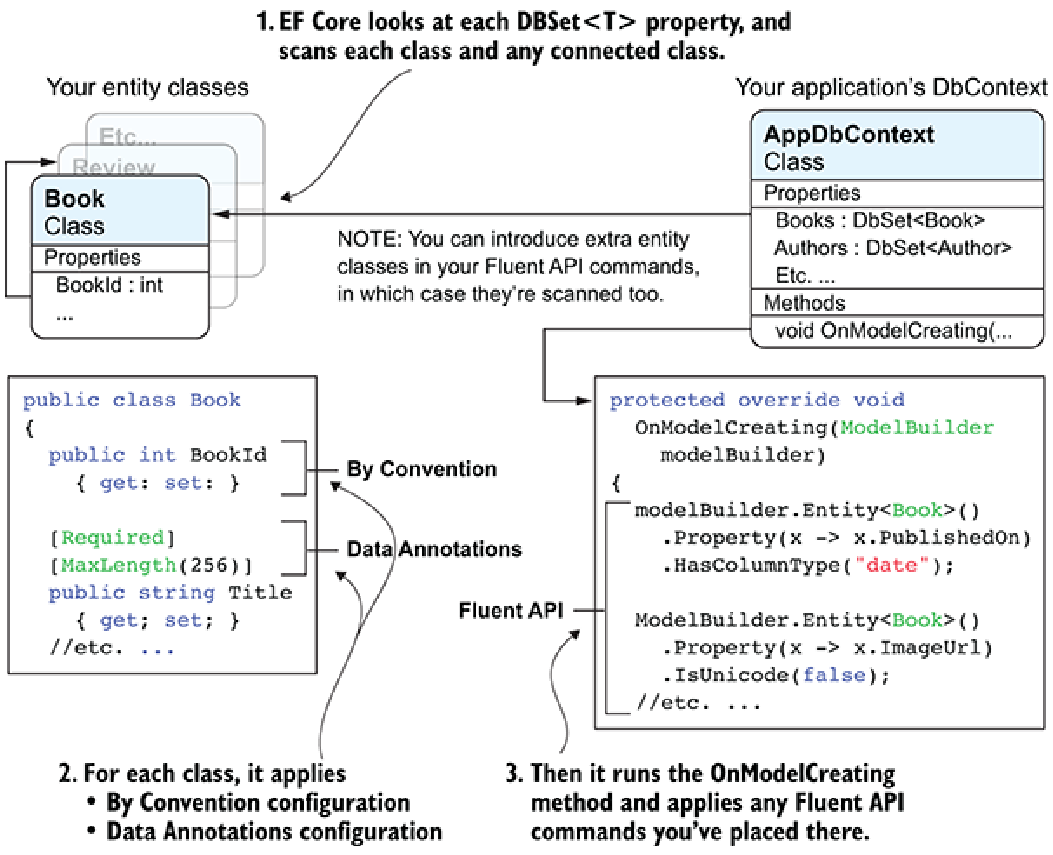
```
class Sales {
	public int SellingPrice;
	public string Name { get; set; }
	public bool IsRegistered { get; }
}
```

CODE 3.1 Plain old CLR object (POCO Classes in Entity FrameWork).

To make this POCO any use for the EF Core it needs to be configured properly.

Conventions are prefined usage of the C# functionalities as they are (Conventions in Entity Framework Core). You just need to know which part of the application EF Core will take as a convention. Data annotations are like keywords but inside square brackets. They add extra meaning to the data by adding attribute tags (Basic Introduction to Data Annotation in .Net Framework). Also the location of data annotation is significant. Lastly there is Fluent API which in EF Core is an object called ModelBuilder. It configures the context with LINQ sequences.

In the graph 3.1 you can see a summary of these three ways of configuration.



1. EF Core looks at each DBSet<T> property, and scans each class and any connected class.

NOTE: You can introduce extra entity classes in your Fluent API commands, in which case they're scanned too.

```
public class Book
{
  public int BookId
    { get: set: }

  [Required]
  [MaxLength(256)]
  public string Title
    { get; set; }
  //etc. ...
```

```
protected override void
  OnModelCreating(ModelBuilder
    modelBuilder)
{
  modelBuilder.Entity<Book>()
    .Property(x -> x.PublishedOn)
    .HasColumnType("date");

  ModelBuilder.Entity<Book>()
    .Property(x -> x.ImageUrl)
    .IsUnicode(false);
  //etc. ...
```

2. For each class, it applies
  - By Convention configuration
  - Data Annotations configuration

3. Then it runs the OnModelCreating method and applies any Fluent API commands you've placed there.

GRAPH 3.1 Configuring an entity (Entity Framework Core in Action, page 148, Figure 6.1).

A good example of these would be a configuration of a primary key column to a table since you can do it with any of the techniques above. There is in Fluent API a *HasKey* method that can handle it (The Fluent API HasKey Method). With annotation you get same results with *Key Attribute* and finally EF Core conventions read *int ClassId* and *int Id* as a primary key as well.

One important thing in these three ways of configuring is that they can all exists alongside with each other and many times they have to since some things which you can do with Fluent API cannot be done with the convention. This in my opinion is very unfortunate since it spreads the configurations around the source code needlessly.

3.2.3 DbContext

Context is a link or session between API and the database. The functionalities of the context are brought to the solution with object deriving from DbContext class exclusively offered by the EF Core. It provides many different functionalities to model building, data mapping, object caching, transaction management, database connections and various data operations (The Entity Framework Core DbContext. Internet article).

From the point of the backend design Context is the place where you configure the way you want the interactions with the database to be formed. If everything goes right you'll never need to write a single query to the database directly but to use your context configuration to design it the way you want and add changes via migrations (3.3.3 Migrations).

DbContext is configured with three types of elements those being DbSet and DbQuery properties and OnModelCreating method. Context maps the POCO classes (3.2.2 Entities) from the model as DbSet types of object and those are further mapped to database as tables. DbQueries are just like DbSets but instead of tables they represent plain SQL-queries.

OnModelCreating method is trickier. It is run after the Context is created the first time and loaded then to the memory. This initializes the model and mappings configured there. Fluent API discussed earlier in section 3.2.2 means in EF Core the chained statements inside the OnModelCreating method keeping the configuration code easier to read and understand while reducing the amount of lines of code (Fluent API Configuration).

3.2.4 Code-first approach

EF Core offers tools to engineer both a backend application model out of database schema or vice versa from source code a database schema. Code-first approach is the latter of those. If you are about to start a project with EF Core from the scratch, code-first takes away all the need of build database yourself. In the best case scenario you don’t ever need to write single line of SQL.

In order to make code-first engineering work your Entity classes needs to be specifically structured according to the requirements of EF Core. These classes need to meet some configuration requirements explained in Entities section 3.2.2 in order to map correctly into tables in database.

3.2.5 Scaffolding

Scaffolding, also known as Reverse-Engineering, is the opposite of the code-first (Reverse Engineering). Scaffolding is a tools-of-choice in case you have already a database and you need to build ORM. This might become the very handy in the situation when old ORM needs to be updated to EF Core. I found scaffolding utmost useful when generating the entity POCOs. Without scaffolding autogeneration functionality I would have needed to code all of the classes one by one manually for each and every table there was in the database.

Scaffolding also includes Fluent API (4.2. Application Programming Interface) and other model configurations the way that after scaffolding it should be possible to access a database with EF Core methods.

Scaffolding also offers some modification possibilities. User can specify the names of the tables and DbContext as well as the output directory and namespace.

This part of the EF Core is still far from finished and it contains some limitations still. Scaffolding is using database schema as a basis for the entity classes it generates. Problem is that not everything is presented in the schema and thus all the configuration outside it will not be passed to the EF Core model. Also some column types are not supported which means that they are not processed at all. Scaffolding also supports only one concurrency token type, row version, so the other won't be passed to the model (Reverse Engineering).

Overall this part of EF Core is still in its development level and making a good code generator is pretty hard after all. Still it's very easy to use and saves a lot of time even though you would need to manually customize the model afterwards.

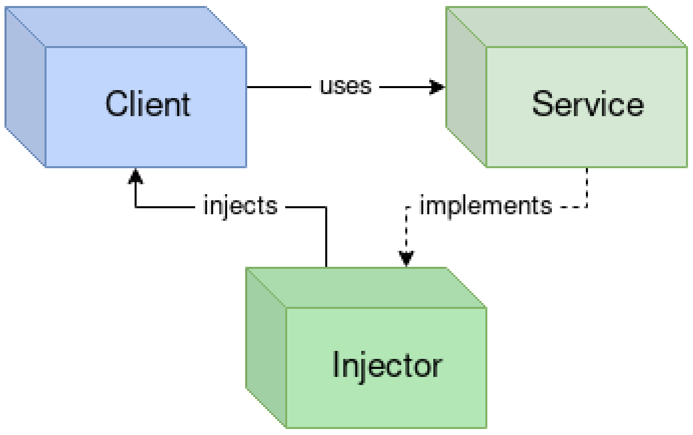
3.3. Advanced concepts

EF Core offers a variety advanced tools when it comes to improving the performance of the application, control over the database behavior or structuring the software. In addition to this ASP.NET Core is offering full support on dependency injection which is the topic in the next chapter.

3.3.1 Dependency injection

Dependency injection (DI) is doing exactly the thing it is saying: injecting dependency. This is especially helpful when some parts of the program require some other parts to be accessible by them in order to work. Normally you would need to build complicated, hard-to-maintain and easy-to-break jungle of dependencies in order to make this happen but with dependency injection only a single passing of parameter to a function is enough (Why does one use dependency injection).

As shown in graph 3.2 dependency injection structures between three parts: Client, Service and Injector classes. While client needs functionalities located in service in order to work, injector is created implementing service and passed to client and thus providing it with everything inside the service class.



GRAPH 3.2 Relationships between the classes in dependency injection (Inversion of control - Dependency injection)

Client can be imagined as a business logic of the application while service is a class that contains some extra functionalities needed by the client but which are not part of business logic. Injector works as a helper class to link these two together in orthogonal manner.

DI can be configured in three different ways: in constructor, method or property. Constructor injection means that the class or interface desired to be injected is passed into an instance as a parameter of a constructor. Method injection means that dependency is injected in a dedicated method instead of straight to constructor. And finally Property injection means that dependency is in public property from which it can be used all around the class. In Shop Management Tool constructor injection used widely

```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _dataAccess;

    public        CustomerBusinessLogic(ICustomerDataAccess
custDataAccess)
    {
        _dataAccess = custDataAccess;
    }

    public CustomerBusinessLogic()
    {
        _dataAccess = new CustomerDataAccess();
    }

    public string ProcessCustomerData(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public interface ICustomerDataAccess
{
    string GetCustomerData(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess()
    {
    }

    public string GetCustomerName(int id)
    {
        return "Dummy Customer Name";
    }
}
```

CODE 3.2 Constructor injection (Inversion of control - Dependency injection)

Use case for dependency injection would be when controller needs an instance of context (3.2.3) to it to establish connection to the database. The only way to make sure the context is instantiated is to inject a dependency to the controller class. In Shop Management Tool this was made via constructor injection.

3.3.2 Loading related data

There's three ways to include external tables to the query with EF Core: eager loading, explicit loading and lazy loading (Loading related data). Eager loading is the simplest of these by loading every time all the relations included into the query (code 3.3.).

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
        .ToList();
}
```
CODE 3.3. Eager loading multiple levels: Blogs -> Posts -> Author (Loading related data. Microsoft documentation).

Explicit loading is more precise in what is loaded. EF Core offers property Entry which can be given the exact row or rows from the database. Then only under that row the related data is loaded. Loaded data can be even after the fetching be filtered further with LINQ as shown in code 3.3.

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

    var goodPosts = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Rating > 3)
        .ToList();
}
```
CODE 3.4. With explicit loading you can specify exactly what to save into memory (Loading related data. Microsoft documentation).

Last and the most sophisticated way of loading data is lazy loading. While other ways of loading uses some EF Core methods or middle part objects like Entry, lazy loading is done by configuring it's own proxy system which handles loading only when it is needed (Lazy loading with EF Core). EF Core makes this easy by offering Microsoft.EntityFrameworkCore.Proxies package. With that lazy loading proxies needs to be configured in at the startup of the application using method UseLazyLoadingProxies (code 3.5.) or with dependency injection passing LazyLoader object to the entity class.

```
private ServiceProvider GetServiceProvider() =>
  new ServiceCollection()
    .AddLogging(config =>
    {
      config.AddConsole()
        .AddDebug()
        .AddFilter(level => level > LogLevel.Debug);
    })
    .AddTransient<BooksService>()
    .AddDbContext<BooksContext>(options =>
    {
      options.UseLazyLoadingProxies()
        .UseSqlServer(Configuration
                                                            .GetConnec-
tionString(BooksConnection));
    })
    .BuildServiceProvider();
```

CODE 3.5 Configuring lazy loading proxies in the startup of application (Lazy loading with EF Core)

There is no one and the best way to load related data. All the the above types are good in different situation. For example if most of the time you need all of the columns from the external table the best choice is eager loading. Explicit loading on the other hand is the loading-type-of-choice when only one column is needed from the external table.

To specify now the relations between tables this can be done in the Fluent API (4.2. Application Programming Interface) or in the domain classes by adding the entity types desired to be loaded as an virtual property under that class.

3.3.3 Migrations

Migration is a collection of changes. In source code it consists of couple of configuration files and in database a one row in a specific migration history table. When developing Shop Management Tool I noticed that each migration is identified by name and timestamp. When migration is referred to in a command it identifies to its name but the order in which migrations are run is sorted by timestamp.

For this reason it is important to make sure that the adjacent migrations are created chronologically in correct order as well.

Migration configuration in source code is separated to a main migration file and designer file (Migration in Entity Framework Core). The main file contains logic necessary to apply the changes. These are formed to Up and Down methods the way that Up contains logic which synchronizes the database with the new migration. Down is used for the reverting changes. In code 3.2 there is a example of this.

```
protected    override    void    Up(MigrationBuilder    migra-
tionBuilder)
{
      migrationBuilder.AddColumn<int>(
            name: "Product",
            table: "Products");
}

protected    override    void    Down(MigrationBuilder    migra-
tionBuilder)
{
      migrationBuilder.DropColumn(
            name: "Product",
            table: "Products");
}
```

CODE 3.2 Migration to add and remove Product column (from Shop Management Tool source code)

Designer file is more like a metadata file for EF Core. It presents the whole model configuration in the form of Fluent API (4.2. Application Programming Interface) in opposition to the user made configuration which can be all around the domain classes or context file.

There is one more configuration file connected to the migrations which is not migration specific but more like keeps the information about the latest model. If the latest migration was done only with the EF Core migration handler script thus without any manual changes, snapshot and the latest migration’s designer file are identical (Entity Framework Core Snapshot).

In addition to these various configurations in the source code EF Core creates also _EFMigrationsHistory table into the database by default. The purpose of this is to keep track which migrations are applied to the database currently adding one extra mean of reassurance for keeping the coherence between backend application and database.

4 STRUCTURE

Architecture choices and the universal ”best practices” which form the structure of application are as important as the technical side of the framework or library you are using. Especially when it comes to the maintainability and refactoring. More about that in this chapter.

4.1. Model-Controller-View

Model-Controller-View or more commonly just MVC is way to structure application. Its main idea is to divide program into three sections and separate their functionalities (MVC Architecture). The separation is following: view takes care of the user interface, controller is responsible of providing view with needed data as well as collecting data from UI and model takes care of data structure. Each one of these are responsible of their own part of the overall data flow.

4.1.1 Model

Model is the data structure of the application. Model’s purpose is to make data available for the rest of the application regardless of where and how it is stored. Whenever data needs to be stored or read from the database the direct calls and queries are done via model (ASP.NET MVC - Model).

In Shop Management Tool model is the place where Entities, Context and their configurations are located. Also DTOs and enumerations are stored in the model.

4.1.2 Controller

Controller controls the data. It is the first point that the HTTP requests will end up to. Controller will handle orderly processing of the external requests whether it needs to change, add or modify the database assets (ASP.NET MVC - Controllers).

In Shop Management Tool controller part can be considered to be the API Server (4.2. Application Programming Interface). In EF Core controller designing has been made fairly easy since every HTTP endpoint is handled via method property under controller class and HTTP request type can be specified by data annotation (Controller methods and views in ASP.NET Core). Actual logic can be written in C#.

4.1.3 View

Almost every application has dedicated user interface. In MVC structure this part is reserved for the View. View is responsible of representing the data and collecting input from the user and funneling it further into the backend (View (MVC)). Unfortunately I cannot go into any further detail because the View, being in part of front-end, is out of the scope of this thesis.

4.2. Application Programming Interface

Application Programing Interface or API for shorter handles communication between two applications. It is the layer that is designed to set the rules how an application is supposed to interact with other applications (What is an API).

One quite handy way to handle application's client-server communication is to set up RESTful API (RESTful API). It is specifically made to handle HTTP-calls. These would be GET, PUT, POST and DELETE. In ASP.NET RESTful API is formed from separated methods which all handle one type of HTTP-call. For example GET-call to http://domain.com/products/1234 will be directed to a method which returns a product with identifier ”1234” while DELETE-call with same URL goes to another method which deletes the same product.

Another important API used in Shop Management Tool is Fluent API (Fluent API in Entity Framework Core). Fluent API's concept of chaining configuration is in the heart of EF Core since it is one way to configure the access to the database. In Code 5.1. is introduced a example of Fluent API configuration for ProductInfo-entity.

```
b.Entity<ProductInfo>(x => {
      x.Property<bool>("IsDeleted").HasDefaultValue(false);
      x.Property<DateTime>("DateAdded")
            .ValueGeneratedOnAdd()
            .HasDefaultValueSql("CURRENT_TIMESTAMP");
      x.Property<DateTime>("LastModified")
            .ValueGeneratedOnAddOrUpdate()
            .HasDefaultValueSql("CURRENT_TIMESTAMP");
});
```

CODE 6.1 Configurations for ProductInfo entity made in Fluent API (source code from Shop Management Tool)

This way the configurations are chained and the one property can be attached to multiple methods so that the whole configuration can be done in single one-liner. This obviously reduces amount of code but might make it harder to read.

4.3. Object-Relational Mapper

Object-Relational Mapper or ORM for shorter is meant to offer way to interact with the database via program code. Traditionally this was reserved for SQL queries (code 6.2.) but ORM made it possible to write queries potentially in every language. Find an ORM example in code 6.3. Nowadays there is ORM for most of the popular programming languages. This means that developers do not need to be masters in SQL anymore in order to write working application with database (What is an ORM and Why You Should Use it).

```
String sql = "SELECT id, name, price, quantity FROM products
WHERE id = 10";
Result res = db.execSql(sql);
String price = res["price"];
```

CODE 4.2 Query returning firstname of person with id 10 with database engine (pseudo code)

In application structure ORM could be thought as a virtualization of the database in the given language (What is Object/Relational Mapping) thus it is somewhere between business logic and database. The main job of ORM is to map the application's objects to structure that the database can understand. In a way it works as translator between SQL and other programming languages. In the code 4.2.

and 4.3. you can see the difference between query made with ORM and the one with database engine.

```
Product p = this.repository.GetProduct(10);
int Price = p.GetPrice();
```
CODE 4.3. Same as in code 4.2. but done with an ORM (pseudo code)

Even though backend contains a lot more than just CRUD functionalities (3.2.1. CRUD) EF Core is in itself just an ORM. Concepts like repository pattern and MVC are essential to backend application but are mostly carried out without EF Core.

5 APPLICATION

The idea of backend is to provide frontend with a possibility to access database (I Don’t Speak Your Language: Frontend vs. Backend) to oversimplify it. In reality there is many decisions to be made in backend designing to assure to the seamless data flow and implementation of needed business logic. In the following chapter I’ll go through the decisions I made when creating backend application with EF Core.

5.1. Architecture decisions

Architecture is the frame on which the whole project relies on. All the parts that have some prefined process or structure are included into it. In this section I will go through three topics: Complexity, abstraction and one widely used standard for software architecture in modern web applications. After those there is something about testing.

5.1.1 Handling complexity

Computer software is inherently complex. As one wise man once said: ”The art of programming is the skill of controlling complexity” (Eloquent JavaScript, page 3). Even the smallest programs contain a lot of tiny parts that need to be made play together in the way that the programmer still has some idea what’s going on. Choosing the right amount of abstraction and sticking to good formatting practices this can be achieved. Making complex software simple to understand is a value in itself since it reduces time spent on thinking what goes and where. It’s also easier to make a mistake in complex than in simple implementation. Big systems tend to be more prone to have bugs than smaller ones. This doesn’t need to be so and that’s why there is many good ways to tame the complexity.

One good principle to make you software easier to approach is called separation of concerns (Separation of concerns). It means that all the parts of software carry out tasks as independently as possible. For example there’s one part to handle the connection to database and another for business logic. This way the business

logic developer doesn’t need to think about data accessing logic and data accessing developer doesn’t need to think about business logic.
Separation of concerns shows in Shop Management Tool by separating the business logic of different entities behind different methods and classes the way that Product entity has ProductController and Sale entity SaleController. Thus the code is more readable and maintainability higher when you don’t need to worry about the entities that are already there.

Coherent naming convention is another crucial part to simplify your program’s source code. Every coding language and framework has certain universally agreed naming conventions which state how you should name your components like types and variables. When it comes to EF Core naming convention of .NET Framework is applied (Naming Guidelines). Furthermore one good practice is to have one class in one file while naming the file with the class name. This makes finding classes easier in IDE’s project explorer.

When it comes to a project folder it is a good idea to introduce some kind of folder structure. In Shop Management Tool this was carried out by separating model into one folder containing the entity classes and the API server as the other folder consisting actual functionalities like REST interface implementations and application startup configurations.

5.1.2 Level of abstraction

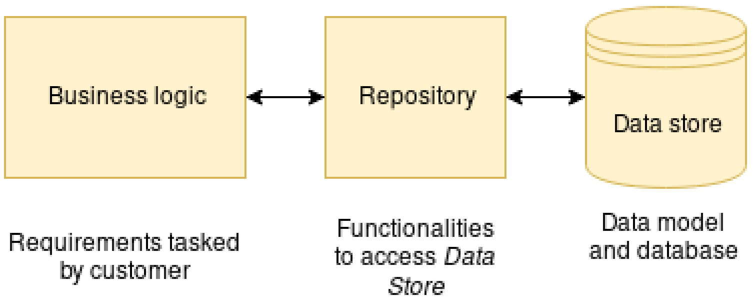
There is many good things in abstraction. With higher level of abstraction software is easier to reuse for different purposes. This will increase much valued scalability. The same high level logic could be used in different parts of the software when formed properly.

Place where this abstraction comes handy is when you are handling almost the same kind of data but not quite enough. In EF Core abstraction is used when adding basic CRUD (3.2.1 CRUD) functionalities to the application. This is done with DbSet called generic class and by instantiating it with entity class. That way all the methods under DbSet are passed for the use of the entity and thus not needed to be included in the entity-specific implementation. In other words abstraction is there to save you from writing or using the same code twice.

5.1.3 Repository pattern

Discussion around repository pattern with EF Core have been vivid since the first versions of it. Some say it’s totally must have (Is the Repository pattern useful with Entity Framework) some it doesn’t offer anything what EF Core wouldn’t already have (Repositories On Top UnitOfWork Are Not a Good Idea). But what is this repository pattern in the first place?

Repository pattern is a way of structuring your application into layers which hide parts of functionality behind dedicated repository class or classes (graph 3.1). In the repository is located needed methods to access database for example: *GetProductByName(string name)* will return the product entity which name matches to the given string-type parameter. This way when designing something else than database connection developer doesn’t need to worry about how the data is moving.


Business logic
Repository
Data store
Requirements tasked by customer
Functionalities to access *Data Store*
Data model and database

GRAPH 3.1 Simplified diagram about repository pattern (Design the infrastructure persistence layer. Documentation page)

This makes all perfectly sense when ORM configuration is complicated. But what if ORM is not that complicated anymore. Or more like: how complicated the ORM needs to be to require repository pattern applied. Many say that EF Core is enough simple to make it obsolete and that repository pattern even hides some important parts of EF Core.

I must agree with that. Firstly because I hate all the excess complexity in the software architecture. Repository pattern tends to make you needlessly spend time trying to find where the actual logic is happening. If this is not absolutely necessary I would advise to avoid it. Secondly creating these repositories takes time. Sometimes abstraction is worth it (5.1.2 Level of abstraction) but it's also possible to over-engineer your application. Over-engineering leads to confusion and thus increases the likelihood of bugs. Thirdly layered code is horrible to read. You might have some idea where everything is if the source code follows some naming standard but you still need to jump from file to file to find the important parts. As an alternative to this you could have everything in one layer or even in one file. To me the latter seems like a better option in most of the cases.

EF Core is made to tackle this excess jumping. What needed to be handled with older ORMs by the developer is now taken care by the EF Core under the hood. Also LINQ is a huge help in this since you can with quite a little amount of readable code create logical queries. This paired with EF Core's CRUD functions creates powerful tool to access database without need of some excess layers in between (code 3.1).

```
var products = this.Context.Products
         .Where(p => !p.IsDeleted)
         .Join(
            this.Context.ProductInfo,
         p => p.Barcode,
         i => i.Code,
            (product, info) => new {
                  Barcode = product.Barcode,
                  Price = product.Price,
                  VatPercentage = product.VatPercentage,
                  Name = info.Name,
                  Unit = info.Unit,
                  IsMeasured = info.IsMeasured,
                  Exclude = info.IsDeleted
                  }
      )
      .Where(x => !x.Exclude);
```

CODE 3.1 Example query with LINQ and EF Core combined (from Shop Management Tool source code)

In my opinion as good as repository pattern is EF Core has made it obsolete. I would advice to let it go in order to fully harness the possibilities of EF Core.

5.2. Programming

Programming is the ground work of software development. For the end product it matters a lot how the projects programming has been carried out. In this chapter I will go through some of the main parts of it.

5.2.1 Programming environment

Programming environment means the developing tools that is needed to deploy a working software up from the source code. These are minimally text editor, build automation tools and debugger. The most common way to form the environment is to use IDE which have all or most of these in the same software. When it comes to Shop Management Tool for me the IDE-of-choice was Visual Studio Code.

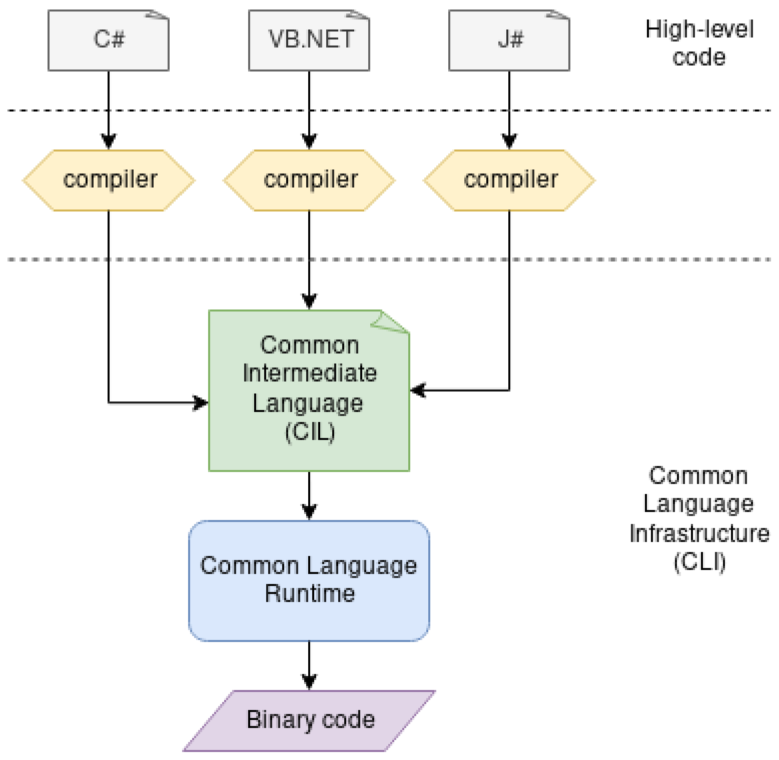
Visual Studio Code is one of the most popular development tools in the world currently (Top IDE Index) due to its simplicity and adaptability. Where other IDEs are inherently made for one programming language or even framework VS Code aims to be modified by extensions offered by the community. If you want to code for example in PHP you just need to download extension for PHP intellisense, color formatting and debugger and you are good to go. The only down side is that the extensions really need to be there and someone needs to support and update them. Some languages (like Java for example) lacks decent extensions in VS Code and thus you still need to pick some other IDE if you want to develop with it.

For me VS Code suited utmost excellently. There was the best tools offered by the .NET Core team itself and I could just choose the ones I needed. This is also one my favorite things in VS Code. For example I didn’t need any test automation added to the IDE so I left it out. There’s also a lot of extra things like code snippets or color themes that might make coding a little bit faster or more pleasant.

One more amazing thing in VS Code is that it works in most of the operating systems including Windows, iOS and most of Linux distributions. A straightforward plugin for terminal proved to be handy especially on Linux since I didn't need keep terminal in separated window as usual but I could run commands in the same window with the source code editor and version control plugin. What's more you can be sure that even if you are need to use some other operating system than the one you are used to, you don't need to change the IDE. VS Code is platform-agnostic.

5.2.2 Compiler

.NET Core is a implementation of Common Language Infrastructure (CLI). CLI contains set of rules and specifications that give a frame for what programming languages need to adhere in order to be fit for the runtime environment (Graph 5.1). In addition to CLI there is a specification called Common Language Specification (CLS) to offer standard for programming languages. The beauty in CLS is that it allows different languages to be compiled to run in the same environment as long as they follow this CLS standard (Common Language Infrastructure (CLI)).



GRAPH 5.1 Place of Common Language Infrastructure in the program code handling process (Overview Of Common Language Infrastructure. Internet article)

Near to CLS fares another standard called Common Type System (CTS). This makes the communication between the different languages (under CLS standard) possible. CTS makes sure that the data types written in different languages can to interact with each other (Common Language Infrastructure (CLI)).

5.2.3 Testing

Testing wasn’t high on my priority at the beginning especially when it came to unit tests. Many people are trying give professional impression by unit testing everything but in my opinion it is just non-sense. Unit tests should be used only in very special places such as when some complicated logic is applied and only when you are writing it yourself. There is no use to test functions offered by gigantic IT companies like Microsoft since they have been most certainly unit tested already by not only the creators but many other developers already. But make no mistake, writing unit tests can be a good idea. You just need to know what you are doing.

Shop Management Tool wasn't created without unit tests. At some point there was fully functional testing structure with mocked repositories xUnit-library to offer some handy functions. Firstly that proved to be a lot of work to set up. Secondly it was quite useless. There just wasn't that many functionalities that unit tests would really benefit since the sole purpose of the backend application was to handle data with methods introduced by EF Core. After a while I ceased writing these useless tests and concentrated on something useful.

At some point I thought the repositories were essential to keep up the unit test structure but actually mocking DbSets (3.2.3 DbContext) is totally enough and thus that's just another reason why EF Core users should not use repository pattern (5.1.3 repository pattern).

Unit tests aren't the only tests there is. For this project the more significant thing was to have a testing side which to use before pushing a new version or feature to production. The basic idea was to have a test database with test live application to play with instead of testing in production but I won't go into any further details on the technical side of it now since it is out of the scope of this thesis.

Anyhow having a test environment is a good idea for many reasons: firstly you can test your application even when the production application is in use and secondly when you test with production environment something can go wrong and in worst case you can break something. And tester should never be afraid of breaking the software when testing. There was many some cases that I introduced some feature to the customer on the test side before actually pushing it to production. This was handy tool to make sure the feature was really the one they wanted.

5.2.4 Customer communication

Very few software projects have too much of customer communication. Same goes with Shop Management Tool. The beginning of the development process was very busy. It was for the customer as well as for me. Neither of us really had the time to arrange decent meetings. The overall situation at the beginning of the project was somewhat chaotic.

Once the most pressing requirements had been deployed and worst bugs solved we had finally some time to organize well-thought meetings. Every meeting had to have some preset agenda and it needed to produce an idea what to do and when. This helped all sides to get bigger picture of the project and thus better understanding what the current status of project is. This way customer has better control on the project and developers end up less frequently in situations where they are creating something that is not needed.

Some notable progress was happening in the course of the project not only the developer side but in the customer as well. Big part of the customer communication is to know how much and with what language to describe the software specifications to the customer who in most of the cases has very little experience on the software development itself. I was quite fortunate to have quick learning and analytically capable representative on the customer’s side so that during these several months of communication and discussion the specifications got better and better.

Something I would advise everyone to aim at is constant learning together with the customer. Achieving this is easier said than done. In my case it took many months to arrive in the point that we could speak about concepts like database. It also saves a lot of time when customer learns what tasks are easy to do software-wise and which ones takes more effort. Same goes the other way around. For programmer it is crucial to understand the business you are optimizing.

6 DISCUSSION

For the purpose of average sized web application Entity Framework Core is more than sufficient. It offered a lot of well-thought out-of-box functionalities needed to build seamless communication between database and API server. Model was easy to create and maintain via migrations and DbContext while LINQ integrated CRUD abilities made it seamless to query database without writing a line of SQL.

Even though Entity Framework Core left me mostly satisfied, some of the features weren’t still complete. Code-generation in scaffolding and migrations were seemingly unfinished and needed after all some manual refining to make them work as intended. One minor inconvenient thing also was a wide variety in types how to configure the model. Instead of just choosing one type and use it throughout the configuration they were configuration specific. This made the code a little bit messy to read and maintain.

Overall my experience is still positive. Setting up the initial application was mostly automated, online documentation was very good and cross-platform support made it flexible to deploy. I would recommend Entity Framework Core to any-one trying to achieve the same goals as I did in this thesis.

REFERENCES

Smith, Jon P., Manning Publications Co., 2018. Entity Framework Core in Action.

Haverbeke, Marijn, No Starch Press Inc., 2019. Eloquent JavaScript, 3rd Edition.

What is Entity Framework? Internet article. Read 10.11.2019. https://www.enti-tyframeworktutorial.net/what-is-entityframework.aspx

.NET Core is the future of .NET. Blog post. Read 12.11.2019. https://devblogs.microsoft.com/dotnet/net-core-is-the-future-of-net/

Loading related data. Documentation page. Read 13.11.2019. https://docs.mi-crosoft.com/en-us/ef/core/querying/related-data

Lazy loading with EF Core. Internet article. Read 9.11.2019. https://csharp.chris-tiannagel.com/2019/01/30/lazyloading/

POCO Classes in Entity FrameWork. Internet article. Read 5.10.2019. https://www.c-sharpcorner.com/UploadFile/5d065a/poco-classes-in-entity-framework/

Is the Repository pattern useful with Entity Framework? Blog Post. Read 23.10.2019. https://www.thereformedprogrammer.net/is-the-repository-pattern-useful-with-entity-framework/

Is the Repository pattern useful with Entity Framework? – part 2. Blog post. Read 23.10.2019. https://www.thereformedprogrammer.net/is-the-repository-pattern-useful-with-entity-framework-part-2/

Repositories On Top UnitOfWork Are Not a Good Idea. Blog post. Read 24.10.2019. https://rob.conery.io/2014/03/03/repositories-and-unitofwork-are-not-a-good-idea/

Overview of the .NET Framework. Documentation page. Read 2.11.2019. https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview?redi-rectedfrom=MSDN

Inversion of control - Dependency injection. Internet article. Read 5.11.2019. https://www.tutorialsteacher.com/ioc/dependency-injection

What is an ORM and Why You Should Use it. Internet article. Read 12.11.2019. https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a

What is Object/Relational Mapping? Internet article. Read 12.11.2019. http://hi-bernate.org/orm/what-is-an-orm/

Why does one use dependency injection? Internet forum post. Read 6.11.2019. https://stackoverflow.com/a/14301496/12304870

Compare EF Core & EF6. Documentation page. Read 14.10.2019. https://docs.microsoft.com/en-us/ef/efcore-and-ef6/

Should I use EF6 or EF Core? Documentation page. Read 16.10.2019. https://docs.microsoft.com/en-us/ef/ef6/#should-i-use-ef6-or-ef-core

Object Relational Mapping Concepts. Internet article. Read 4.11.2019. https://medium.com/@codeshifu/object-relational-mapping-concepts-e2ff0838590c

What is CRUD? Internet article. Internet article. Read 15.10.2019. https://www.codecademy.com/articles/what-is-crud

Basic Introduction to Data Annotation in .Net Framework. Internet article. Read 18.10.2019. https://code.msdn.microsoft.com/Basic-Introduction-to-Data-244734a4

The Entity Framework Core Model. Internet article. Read 18.10.2019. https://docs.microsoft.com/en-us/ef/ef6/#should-i-use-ef6-or-ef-core

The Fluent API HasKey Method. Internet article. Read 11.11.2019. https://docs.microsoft.com/en-us/ef/ef6/#should-i-use-ef6-or-ef-core

The Entity Framework Core DbContext. Internet article. Read 10.11.2019. https://www.learnentityframeworkcore.com/dbcontext

Fluent API Configuration. Internet article. Read 7.11.2019. https://www.learnenti-tyframeworkcore.com/configuration/fluent-api

Reverse Engineering. Documentation page. Read 3.11.2019. https://docs.micro-soft.com/en-us/ef/core/managing-schemas/scaffolding

Migration in Entity Framework Core. Internet article. Read 19.10.2019. https://www.entityframeworktutorial.net/efcore/entity-framework-core-migra-tion.aspx

Entity Framework Core Snapshot. Internet article. Read 19.10.2019. https://softdevpractice.com/blog/entity-framework-core-snapshot/

Naming Guidelines. Documentation page. Read 11.10.2019. https://docs.micro-soft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines

Overview Of Common Language Infrastructure. Internet article. Read 14.11.2019. https://www.c-sharpcorner.com/blogs/overview-of-common-lan-guage-infrastructure

Common Language Infrastructure (CLI). Internet article. Read 30.10.2019. https://searchapparchitecture.techtarget.com/definition/Common-Language-Infrastructure-CLI

MVC Architecture. Internet article. Read 29.10.2019. https://www.tutorialsteacher.com/mvc/mvc-architecture

ASP.NET MVC - Model. Internet article. Read 12.10.2019. https://www.tutorialsteacher.com/mvc/mvc-model

ASP.NET MVC - Controllers Internet article. Read 16.10.2019. https://www.tutorialspoint.com/asp.net_mvc/asp.net_mvc_controllers.htm

Controller methods and views in ASP.NET Core. Documentation page. Read 21.10.2019. https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/controller-methods-views?view=aspnetcore-3.0

View (MVC). Internet article. https://www.techopedia.com/definition/27453/view-mvc-aspnet

Fluent API in Entity Framework Core. Internet article. Read 24.10.2019. https://www.entityframeworktutorial.net/efcore/fluent-api-in-entity-framework-core.aspx

What is an API? Internet article. Read 1.11.2019. https://apifriends.com/api-management/what-is-an-api/

RESTful API. Internet article. Read 18.10.2019. https://searchapparchitecture.techtarget.com/definition/RESTful-API

I Don't Speak Your Language: Frontend vs. Backend. Internet article. Read 7.10.2019. https://blog.teamtreehouse.com/i-dont-speak-your-language-frontend-vs-backend

Separation of concerns. Internet article. Read 3.10.2019. https://deviq.com/separation-of-concerns/

Design the infrastructure persistence layer. Documentation page. Read 20.11. 2019. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design

Top IDE Index. Internet article. Read 4.11.2019. https://pypl.github.io/IDE.html

Conventions in Entity Framework Core. Internet article. Read 28.11.2019. https://www.learnentityframeworkcore.com/conventions

APPENDICES

Appendix 1. Entity Framework and Entity Framework Core comparison chart

This chart is comparing versions up to Entity Framework 6.2.0 and Entity Framework Core 3.0.0. (Compare EF Core & EF6. Microsoft documentation.)

Creating a model

Feature	EF 6	EF Core
Basic class mapping	Yes	1
Constructors with parameters		2.1
Property value conversions		2.1
Mapped types with no keys		2.1
Conventions	Yes	1
Custom conventions	Yes	1.0 (partial)
Data annotations	Yes	1
Fluent API	Yes	1
Inheritance: Table per hierarchy (TPH)	Yes	1
Inheritance: Table per type (TPT)	Yes	
Inheritance: Table per concrete class (TPC)	Yes	
Shadow state properties		1
Alternate keys		1
Many-to-many without join entity	Yes	
Key generation: Database	Yes	1
Key generation: Client		1
Complex/owned types	Yes	2
Spatial data	Yes	2.2
Graphical visualization of model	Yes	
Graphical model editor	Yes	
Model format: Code	Yes	1
Model format: EDMX (XML)	Yes	
Create model from database: Command line	Yes	1

Create model from database: VS wizard	Yes	
Update model from database	Partial	
Global query filters		2
Table splitting	Yes	2
Entity splitting	Yes	
Database scalar function mapping	Poor	2
Field mapping		1.1
Nullable reference types (C# 8.0)		3

Querying Data

LINQ queries	Yes	1.0 (in-progress for complex queries)
Readable generated SQL	Poor	1
GroupBy translation	Yes	2.1
Loading related data: Eager	Yes	1
Loading related data: Eager loading for derived types		2.1
Loading related data: Lazy	Yes	2.1
Loading related data: Explicit	Yes	1.1
Raw SQL queries: Entity types	Yes	1
Raw SQL queries: Keyless entity types	Yes	2.1
Raw SQL queries: Composing with LINQ		1
Explicitly compiled queries	Poor	2
Text-based query language (Entity SQL)	Yes	
await foreach (C# 8.0)		3

Saving data

Change tracking: Snapshot	Yes	1

Change tracking: Notification	Yes	1
Change tracking: Proxies	Yes	
Accessing tracked state	Yes	1
Optimistic concurrency	Yes	1
Transactions	Yes	1
Batching of statements		1
Stored procedure mapping	Yes	
Disconnected graph low-level APIs	Poor	1
Disconnected graph End-to-end		1.0 (partial)

Other features

Migrations	Yes	1
Database creation/deletion APIs	Yes	1
Seed data	Yes	2.1
Connection resiliency	Yes	1.1
Lifecycle hooks (events, interception)	Yes	
Simple Logging (Database.Log)	Yes	
DbContext pooling		2

Database providers

SQL Server	Yes	1
MySQL	Yes	1
PostgreSQL	Yes	1
Oracle	Yes	1
SQLite	Yes	1
SQL Server Compact	Yes	1.0
DB2	Yes	1

Firebird	Yes	2
Jet (Microsoft Access)		2.0
Cosmos DB		3
In-memory (for testing)		1