Masaryk University
Faculty of Informatics

Study materials for the web development in .NET Core

Bachelor's Thesis

Viliam Popróči

Brno, Fall 2019

Masaryk University
Faculty of Informatics

Study materials for the web development in .NET Core

Bachelor's Thesis

Viliam Popróči

Brno, Fall 2019

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Viliam Popróči

Advisor: Mgr. Martin Macák

Acknowledgements

I want to thank my supervisor, Mgr. Martin Macák for offering me to work on this thesis. His support, guidance, and patience greatly helped me to finish it. I would also like to thank my consultant, Bc. Štefan Bojnák, who had given a different perspective on the topic.

The thanks and gratitude also belongs to the family and friends for support.

Abstract

The goal of this thesis is to develop a complex web application, which serves as educational material for the PV179 course. The application is running on a relatively new web development framework ASP.NET Core. The first part of the thesis focuses on the main differences of former ASP.NET framework and ASP.NET Core framework. The following part is describing used design patterns. The last part contains the project structure, architecture and the technologies used in it.

Keywords

.NET Core, ASP.NET Core, ASP.NET Core MVC, Web application, design patterns, Three-Tier architecture, study materials

Contents

List of Figures

Introduction

Nowadays, all the information can be found on the internet, consisting of various web applications, which are used in everyday life. Every year there are new technologies that help with almost everything developers do. The developers are always forced to learn new things and techniques. That being said, there is always room to improve.

This thesis tackles the given problem. The primary goal of it is to implement web application, which will serve as study material for the PV179 course in modernized technology C# ASP.NET Core. Currently implemented application is running on an older technology called ASP.NET framework. Even if ASP.NET Framework is not outdated yet, technology is improving at fast rate. There may be no new releases, meaning that one day it will lose compatibility with the future operating systems. However, considering how many new features the new ASP.NET Core is bringing, it would be a mischance, not to take advantage of it.

As mentioned above, the project is a web application implemented in C# ASP.NET Core. The application itself is a content management system - a user-friendly interface for changing the content of the website dynamically. It consists of twenty-two iterations, wherein the first one students are starting from scratch. In the last one, there is a fully functioning application with all the features content management systems should have while emphasizing on the correct usage of known design patterns and architectures.

The first chapter describes the main differences between the mentioned technologies and comparing them to each other. Afterwards, the second chapter focuses primarily on design patterns. The project uses several of them to show why and when is the scenario to use them. The following third chapter analyzes the implementation and structure of the project. It also describes the usage of the third-party technologies and libraries in the project and describes their helpfulness.

The last chapter concludes the thesis with the overall contribution of the project and possible improvements in the future.

1 ASP.NET Core MVC

This chapter contains information about ASP.NET Core and the main differences between this and the older ASP.NET Framework.

1.1 Characteristics and motivation

ASP.NET Core is a framework for developing web applications. It combines the most useful parts from .NET Framework, techniques from agile development and efficiency of model-view-controller (MVC) architecture [1]. The primary motivation to use ASP.NET Core in the application is that it is running on .NET Core, which is similar to .NET Framework, but much more efficient and not Windows dependent version of it.

Although Windows is still a leading operating system in terms of usage, web applications are more popular on smaller and simplified containers in the cloud platform. Microsoft extended its reach of .NET by embracing the cross-platform approach and making it possible to host ASP.NET Core applications in a broader spectrum of hosting environments. Developers now can take advantage of powerful C# applications on other operating systems such as Linux, macOS, Android, and iOS. Moreover, .NET Core is a fully open-source, meaning source codes of the Framework can be downloaded, rewritten, and compiled to the modified version of it, which may come in remarkably handy. The perfect example is running into system component error, debugging it with the privilege to read the commentary from the original developer, or testing the limits of the components in the process of developing more advanced structures.

Besides Visual Studio, which is the most used integrated development environment (IDE) for C#, there is also an open-source, cross-platform lightweight version called Visual Studio Code, which means the development is no longer strictly for Windows.

1.2 Differences

The goal of this section is to pick the essential differences affecting the web application.

1.2.1 Installation

Installation of the .NET Framework requires to be a single package and runtime environment for Windows. However, as mentioned above, .NET Core is cross-platform and needs to be packaged and installed independently of the underlying operating system [2]. All of the NuGet[1] packages and dependencies are compiled and included in the .NET Core applications.

1.2.2 Hosting

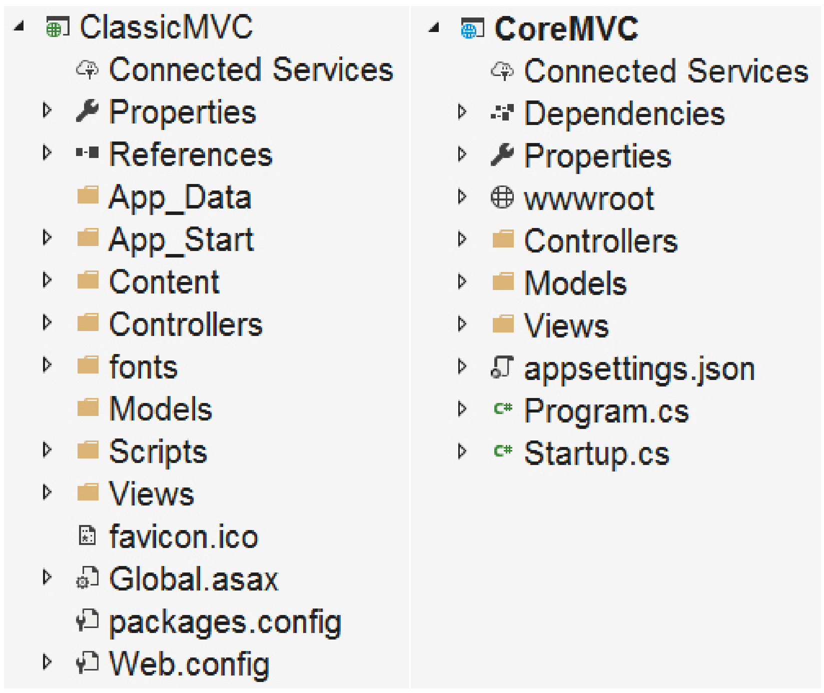
The .NET Framework web applications are bind solely to Internet Information Services (IIS), which is provided by Windows only. On the other hand, .NET Core web applications, as they are cross-platform, can run on software such as Nginx or Apache.

1.2.3 Project structure

There are several significant changes in the project structure in MVC web applications. From the first look at Figure 1.1, it has a more ordered and less complicated appearance, which means it is easier to navigate through.

The first thing is that there no longer is a *References* item, but *Dependencies* instead. The difference between these two may appear insignificant and preferably cosmetic. Although the logic is the same and both serve the same purpose, in .NET Core, there were added dependencies for javascript libraries such as NPM and Bower. Moreover, as mentioned above, the *References* in the .NET Framework are referring to installed packages, whereas *Dependencies* in .NET Core refers to packages inside the application. These packages will appear in the bin/debug folder after building the application.

1. https://www.nuget.org/



```
ClassicMVC
    Connected Services
    Properties
    References
    App_Data
    App_Start
    Content
    Controllers
    fonts
    Models
    Scripts
    Views
    favicon.ico
    Global.asax
    packages.config
    Web.config
```

```
CoreMVC
    Connected Services
    Dependencies
    Properties
    wwwroot
    Controllers
    Models
    Views
    appsettings.json
    Program.cs
    Startup.cs
```

Figure 1.1: Project structure

The simple cosmetic change in the .NET Core web applications is getting rid of folders such as *Scripts*, *App_Data*, *App_Start*, *Fonts*, *Content* and unnecessary static files in the project root. All static files sent to the browser are now in a single container *wwwrooot*. The rest of the folders – *Models*, *Views*, *Controllers* – are the same.

Another thing to notice is missing *web.config*, which was XML file responsible for the configuration of the web application. In the .NET Core, all the configuration goes to *appsettings.json*. The significant difference is that *web.config* is a structured XML file, which is difficult to read and write and often configured trough IIS, telling IIS how to serve the application to the browser. On the other hand, in ASP.NET Core, the application settings are structured in JSON and are very simplified, which makes them rather easy to write and read. Nevertheless hosting a .NET Core web application in Windows environment and IIS still requires to generate simple *web.config*.

The code, in most parts, is the same as far as for controllers and views goes, although there are slight differences. Tag helpers can be used in views, which are doing the same thing, but appear much tidier in terms of semantics. The possibility of using the same razor syntax and HTML helpers is still there. Moreover, the input validation, routing and binding work remarkably better. Although the MVC codebase remains similar or the same, the **Startup.cs** file has some significant changes.

1.2.4 Configuration

Both ASP.NET Core and ASP.NET is running on a technology called middleware. Middleware is application pipeline to handle requests and responses [3]. It can be imagined as a queue of people handing some package to the one behind when the package arrives at the last person; the package is opened and sent back to the front. In the case of web applications, the HyperText Transfer Protocol (HTTP) request is the package as can be seen in Figure 1.2.

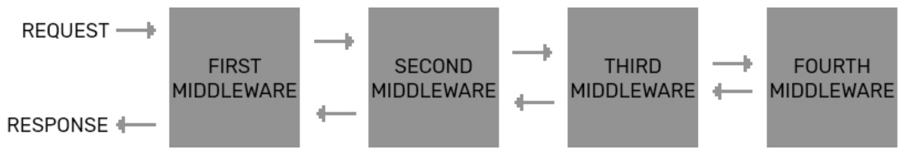


Figure 1.2: Middleware illustration

In other words, the configuration of the application is set by adding, creating, and ordering middlewares in the **Startup.cs** file. However, the most significant change in configuration is that ASP.NET Core has native dependency injection (DI), meaning third-party libraries are not necessary for resolving dependencies. Moreover, injecting is intuitive and testing can be done more effortlessly and efficiently.

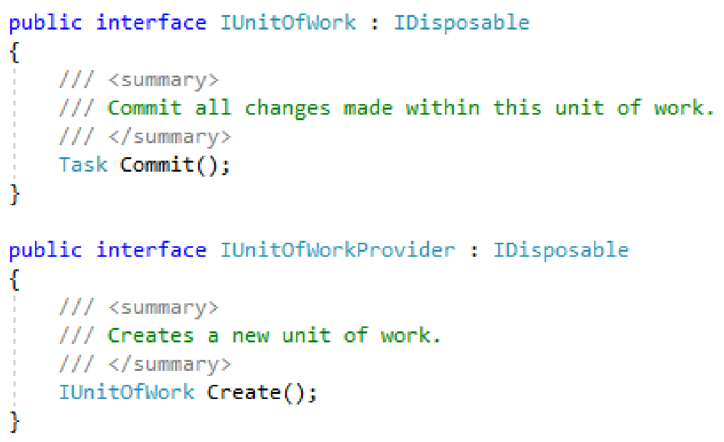
2 Design Patterns

This chapter focuses primarily on design patterns the project is using and following.

2.1 Unit of Work

Unit of work (UoW) is a design pattern with two prime responsibilities. Maintain the information passing to the database, and pass information to the database. It handles these operations as a transaction, so when some error occurs during the process, the database rolls back to the state before the transaction [4].

Splitting UoW to single words of *unit* and *work* can provide better understanding of the pattern. Work in software application can be described as inserting, updating, and deleting data from the database. Whereas the unit stands for a single unit, which observes ongoing processes in a single transaction.



```
public interface IUnitOfWork : IDisposable
{
    /// <summary>
    /// Commit all changes made within this unit of work.
    /// </summary>
    Task Commit();
}

public interface IUnitOfWorkProvider : IDisposable
{
    /// <summary>
    /// Creates a new unit of work.
    /// </summary>
    IUnitOfWork Create();
}
```

Figure 2.1: Unit of work interface

In the project, there is an Unit of Work, which is calling the commit to database, and Unit of Work Provider, which only holds the instance of currently used UoW. The usage is moderately straightforward, and it tackles the problem of undesired data in the database. In Figure 2.1

is the code outline of how the interface looks. At the start of the transaction, the service, which handles the data, is invoking the provider's *Create* method, and at the end of the transaction, the same service is invoking UoW's *Commit*.

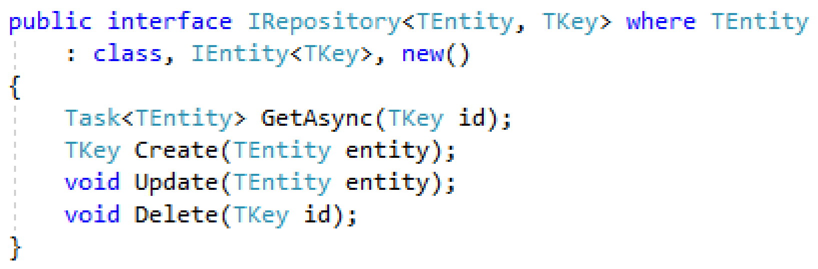
2.2 Repository

The repository is another level of abstraction placed in the project infrastructure (see Figure 3.2). The real purpose of the repository is to create communication between the data access layer and business layer. It gets the entities from the database and maps them to the business entities, with or without ORM (see Section 3.5.1). It behaves like a group of collection methods [5].

The repository only performs create, retrieve, update, delete (CRUD) operations, and there are several approaches on how to implement it. The first and probably the most natural one is to have one repository per business entity. However, this is highly unpractical because of having too much redundant code. The second, trimmed way is to make a generic repository, which is the way the project is working. The project is using the combination of the Repository and Query Object pattern - see the following section. It is common to use the repository pattern without even realizing it because Entity Framework has a built-in implementation of it. However, since the project is using more than one ORM to show how easy it is to swap the data access layer, without changing the infrastructure. A good example is, using several databases and data approaches, but the repository implementation would remain the same. The point is that the repository provides clean access to the entities from which BL can easily read and write to it, without worrying where the data is or under what storage the data is. Figure 2.2 shows how the project repository looks like; the methods are self-explanatory.

2.3 Query object

Query Object is a design pattern prepared to represent a SQL query. Its fundamental purposes are allowing to build queries and to utilise those object structures into the fitting SQL query [6].



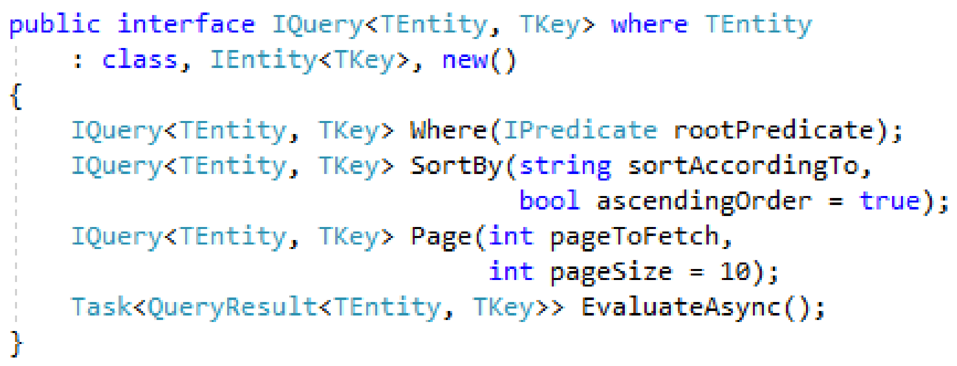
```
public interface IRepository<TEntity, TKey> where TEntity
    : class, IEntity<TKey>, new()
{
    Task<TEntity> GetAsync(TKey id);
    TKey Create(TEntity entity);
    void Update(TEntity entity);
    void Delete(TKey id);
}
```

Figure 2.2: Repository interface

Another part of the infrastructure is the query object, and it has a similar purpose as the repository. The problem of having one to one relation for query object and entity remains. The query object needs to be generic, as well as in the case of the repository.

The primary difference between the repository and the query object is that, the repository is used only for CRUD operations whereas the query object serves as a retriever of the multiple entities of the same type. Using Query Object in the application helps retrieving the data under some conditions, in other words filtering. For example, the application contains multiple images, and the user wants to find only the first ten images ordered by date of creation.

The implementation might be much more complicated for the most effective approach. Implementing the query object throughout Expression trees and reflection might not have a performance the application needs. The project is using a direct approach to the database via SQL. The application is building a resulting SQL query based on user filters and demands, which is then evaluating in the database. However, even if it is the fastest approach, it is of use strictly in the relational databases. Figure 2.3 shows the Query interface of the project.

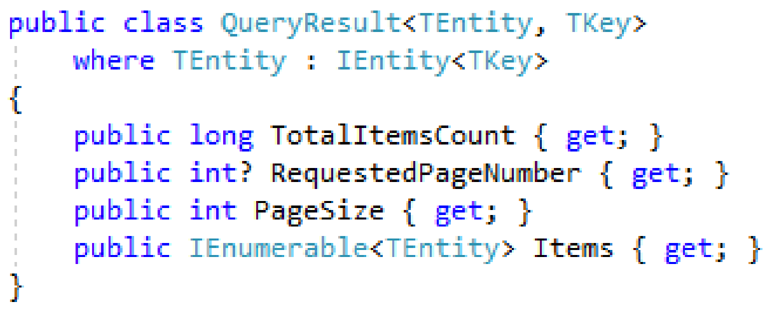


```
public interface IQuery<TEntity, TKey> where TEntity
    : class, IEntity<TKey>, new()
{
    IQuery<TEntity, TKey> Where(IPredicate rootPredicate);
    IQuery<TEntity, TKey> SortBy(string sortAccordingTo,
                               bool ascendingOrder = true);
    IQuery<TEntity, TKey> Page(int pageToFetch,
                             int pageSize = 10);
    Task<QueryResult<TEntity, TKey>> EvaluateAsync();
}
```

Figure 2.3: Query interface

The *IPredicate* is an interface for the predicate structure the project is using. There are two types of predicates. The **SimplePredicate** and the **CompositePredicate**. The simple predicate is a single basic condition with value comparing operator and actual value. The composite predicate is an array of *IPredicates* with a logical operator.

The **QueryResult** is the class that represents the final result of the query (see Figure 2.4).



```
public class QueryResult<TEntity, TKey>
    where TEntity : IEntity<TKey>
{
    public long TotalItemsCount { get; }
    public int? RequestedPageNumber { get; }
    public int PageSize { get; }
    public IEnumerable<TEntity> Items { get; }
}
```

Figure 2.4: QueryResult class

2.4 Service

Service is another level of abstraction above the query objects and repositories [7]. In the project, it provides the mapping of the database entities to Data Transfer Objects (DTOs) (see Section 2.6). Services combine the repositories and the query objects as illustrated in Figure 2.5. In the project, there is a single service for every entity. It may appear that there is excessive amount of code; nevertheless, the implementation is coherent with the use of generics for CRUD operations. The prime reason for having services is that it provides CRUD operations, and additionally calculations and modifications to the entities before they go to the repository or the higher level. For example, it computes the hash and salt for passwords, resizes images, or shrinks documents. It is also a level of abstraction that is independent of the data access layer because repositories and query objects handle all the storing logic.

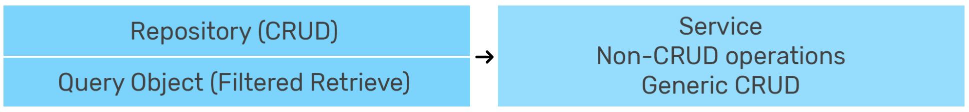


Figure 2.5: Service diagram

2.5 Facade

Facade is an object, which serves as an interface for the presentation layer. It hides complex code and provides simple use of it [8].

In the project, facades combine and merge multiple services and are part of the business layer that provides logic for the presentation layer. Figure 2.6 shows aggregation process in the project.

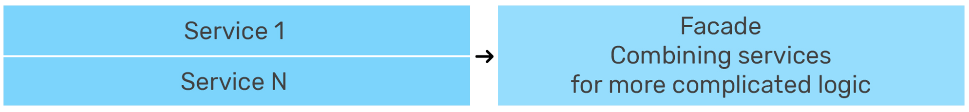


Figure 2.6: Facade diagram

The services are providing facades with needed functionality. By combining them together, more complex solutions can be made in this layer. That allows writing almost zero application logic inside the controllers in the presentation layer. Facades communicate with controllers only via data transfer objects, and therefore the presentation layer has no knowledge of business entities. Moreover, the presentation layer no longer needs any dependencies necessary in the business layer.

2.6 Data Transfer Object

Data Transfer Object (DTO) is the object that carries data between layers. The motivation to use it is to reduce the size of the transferred data, which results in faster client-server communication [4]. Moreover, the application is protecting some data this way, by not sending all fields to the user. For example, retrieving user credentials to the presentation layer is a major problem due to the possibility of causing security breaches. The application should provide only the utterly necessary information for the smooth user experience. DTOs do not contain any behaviour, meaning they only serve one purpose, which is to transfer data. However, they might contain some serialisation logic because REST service often uses them as a returning type. For example, when the application is serving an array of files where only the name and size of the file is necessary. The application should ignore the data of the file and serve what is needed; otherwise, it would be very ineffective.

2.7 Model-View-Controller

MVC stands for Model, View, and Controller. MVC is a preferred way of organising the code in web applications [9]. MVC's biggest advantage is that each part of it has its specific purpose. The model stands for the code that holds application data. The view contains the part that the user sees and interacts. The controller has a purpose of the functionality of the application. In the application, the DTOs retrieved from the facade of BL can be considered the model.

The controller contains all the application logic, which operates the model. The controller receives the model from the underlying facade, and afterwards, it populates it to the view, which contains all the graphics and active elements of the page.(HTML, CSS, JS).

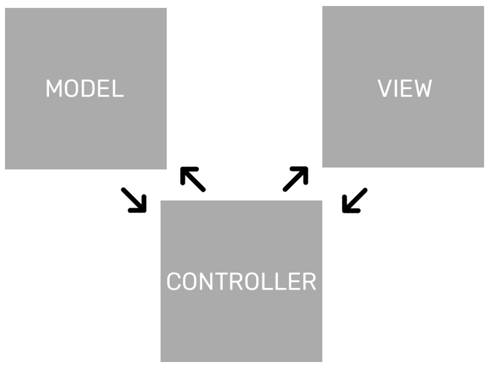


Figure 2.7: MVC flow

Figure 2.7 shows how the flow works. Firstly in the controller, the application gets the model and then populates the view with it. Then the application registers the user's actions in view, processes it through the controller, which updates the model and returns it once again.

3 Project

This chapter contains the details about the project architecture, implementation and structure. Furthermore, it provides information about the technologies and libraries that the project is using.

The project itself is content management system (CMS) [1], which has several functions. It can create and manage the posts, pages, and articles, comment on them, assign them to categories. Control the images, videos, documents, and other files. Moreover, users can register, login and manage the content in the administration section.

3.1 Three-Tier architecture

Three-Tier architecture is a way of application development composed of three layers of logical computing [10]. The main reason for using Three-Tier architecture is to separate the application into data access layer, business layer and presentation layer. Each layer has its specific purpose, and each is dependent strictly on the one below. The simplistic version of the architecture is illustrated in Figure 3.1.

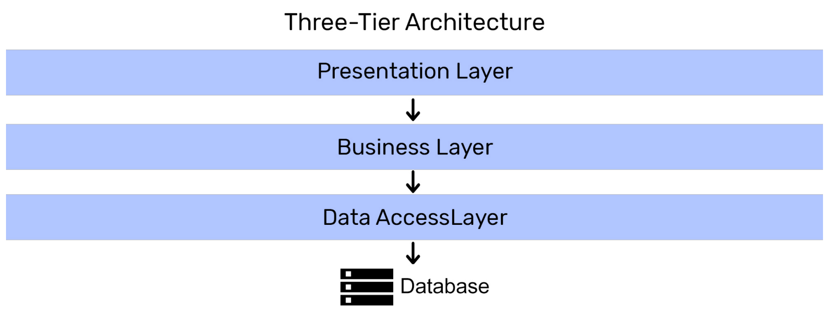


Figure 3.1: Simplified Three-Tier architecture

The data access layer (DAL) serves as the communication with the storage such as the relational databases. Structured Query Language (SQL) procedures or Object-relational mapping (ORM) (see

1. https://searchcontentmanagement.techtarget.com/definition/content-management-system-CMS

Section 3.5.1) is the fundamental part of this particular layer, because it provides simplified access to data in storage.

The business layer (BL) serves as communication with the DAL. It consists of all the logical or numerical computations over the data and provides the results for the upper layer.

At the top is the presentation layer (PL). All components of the user interface (UI) sits in this layer. In web applications, it is mostly Hypertext Markup Language (HTML), Cascading Style Sheet (CSS) and JavaScript (JS). This layer communicates with the BL, which provides the data served to the user.

The main benefits of Three-Tier architecture are the speed of development, scalability, performance, and availability [10]. It is straightforward to split the work between more developers, meaning everyone can focus on their work, and it is beneficial because the changes of some code in one layer have almost zero impact on the other one. The designers and coders can focus strictly on the PL, while programmers can develop the BL, without interfering with each other. In the perfect real-world scenario, there would be a server for each layer. That is the case where scalability comes in. The situation where there are many requests on the BL, rescaling of that concrete server should be enough without any further expenses. That brings another advantage because of managing the resources provided for the development.

On the other hand, it is not always worth to go for Three-Tier architecture. It can be very time-consuming to prepare in a situation, when there is a need to develop a smaller application. It might take more time to make the architecture than to build the application itself.

Figure 3.2 shows how architecture looks in the project.

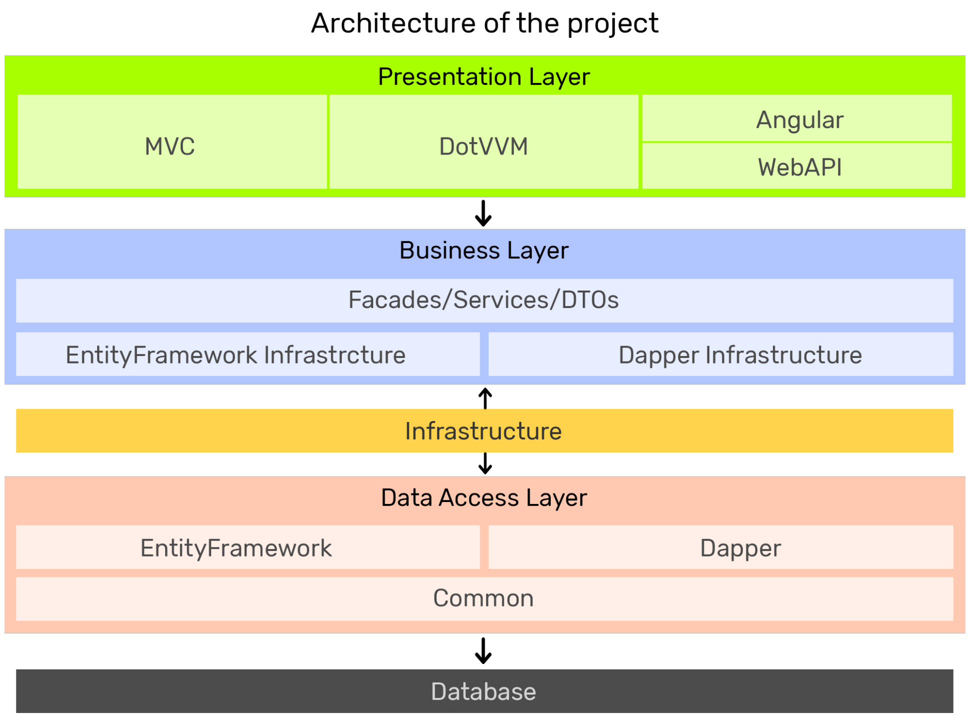


Figure 3.2: Architecture used in the project

3.2 Class Diagram

Although the class diagram in the project is rather simplistic, it demonstrates the most common use cases of CMS with it (see Figure 3.3).

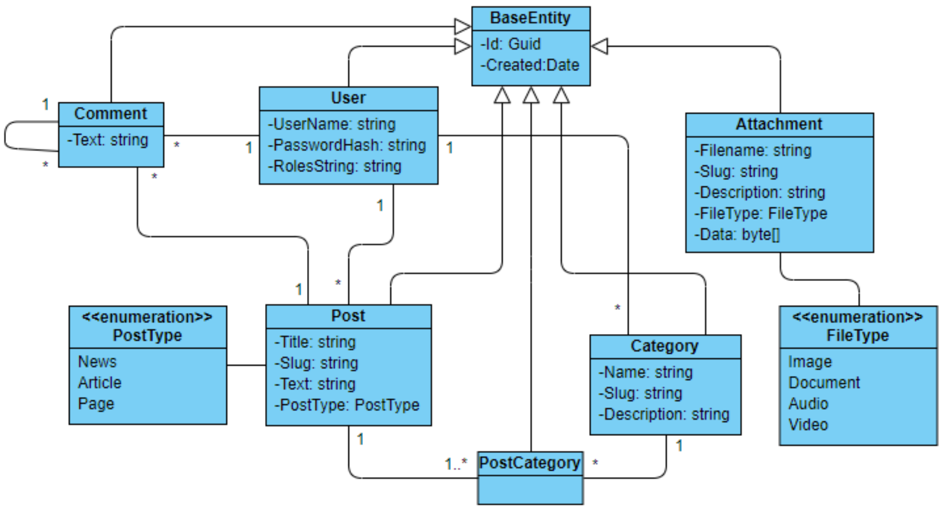


Figure 3.3: Class diagram of the project business entities

3.3 Requirements

The only requirement for this project is to have .NET Core 2.2 SDK[2] and Visual Studio, Raider or Visual Studio Code IDE[3]. The biased recommendation is Visual Studio Code because it is open-source and free.

3.4 Structure

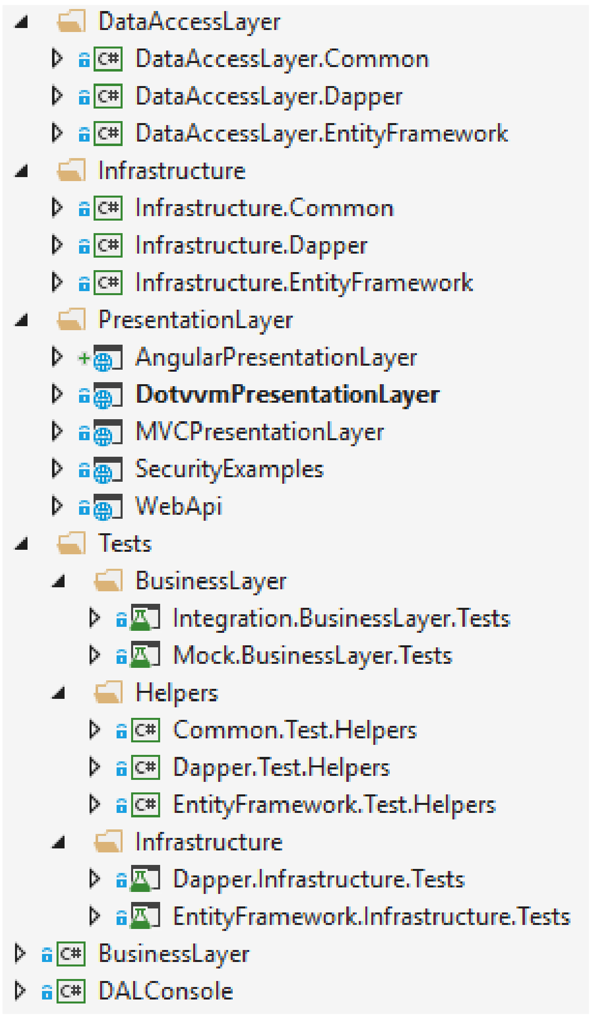
3.4.1 Zip file structure

The zip file contains twenty-two iterations of the project; iterations make tuples of the task and the solution. There are eleven folders called *labX*, where X stands for the number of the given tuple, each of them contains two folders *labX/task* and *labX/finished*. The task folders contain unfinished project iteration and todo list with information for instructors of the course. As name *finished* suggests, the finished folder includes the complete version of given task iteration.

2. https://dotnet.microsoft.com/download
3. https://code.visualstudio.com/

3.4.2 Last solution iteration

The final project is rather complicated, and this section describes the entire solution. Figure 3.4 shows the entire solution structure and all the sub-projects contained in it.



```
DataAccessLayer
    DataAccessLayer.Common
    DataAccessLayer.Dapper
    DataAccessLayer.EntityFramework
Infrastructure
    Infrastructure.Common
    Infrastructure.Dapper
    Infrastructure.EntityFramework
PresentationLayer
    AngularPresentationLayer
    DotvvmPresentationLayer
    MVCPresentationLayer
    SecurityExamples
    WebApi
Tests
    BusinessLayer
        Integration.BusinessLayer.Tests
        Mock.BusinessLayer.Tests
    Helpers
        Common.Test.Helpers
        Dapper.Test.Helpers
        EntityFramework.Test.Helpers
    Infrastructure
        Dapper.Infrastructure.Tests
        EntityFramework.Infrastructure.Tests
BusinessLayer
DALConsole
```

Figure 3.4: Entire solution structure

DataAccessLayer folder

This folder contains three sub-projects, which are only class libraries. The first one, named **Common**, contains the base entity model and enumerated types. The two others contain the implementation of so-called Database Context, which in both cases contains configuration for the real database connection. In Entity Framework, the DB context also contains virtual collections of the data called DbSets.

Infrastructure folder

Another similarly ordered and named projects. The **Common** project contains mainly interfaces for the infrastructure, whereas the rest of them focuses on the real implementations. The infrastructure contains the implementation of Query Object, Repository, and Predicates. The descriptions of query objects, repositories and predicates are in the previous chapter.

BusinessLayer project

All the application logic is in this project, meaning all the services, DTOs, Query Objects, Facades, and mapping configuration. Information to these is in the previous chapter except mapping configuration, which is a file that specifies how business entities should be mapping into DTOs (see Section 3.5.3).

PresentationLayer folder

There are five projects in this folder; the first one is **AngularPresentationLayer**. This project serves only as an example project for introduction to javascript framework called Angular (see Section 3.5.5).

The second project is **DotvvmPresentationLayer**, which contains fully functioning PL based on a framework called DotVVM - 3.5.6.

MVCPresentationLayer, as the name suggests, is the classic complete PL build on classic MVC and Razor Pages. This project is the prime PL.

The **SecurityExamples** is only an example project for security breach lecture. It contains only one controller with prepared exercises

— for example, Denial of Service (DoS)[4] prevention, SQL Injection[5], or Cross-Site scripting (XSS)[6].

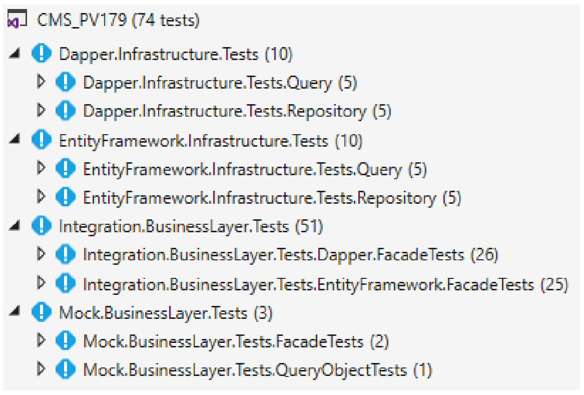
The last project is the **WebApi**, which serves as the Representational State Transfer REST application, and it is necessary to have it running for the Angular project to run correctly.

Tests Folder

In this folder, there are several other folders. At the lowest level, there are three class libraries with Helpers for the actual tests. These contain some overrides, data seeds, and configurations necessary for testing.

The other two folders are self-explanatory. However, the tests in *BusinessLayer* folder the *Integration* tests project contains tests for the facades with both infrastructures, meaning Dapper and Entity Framework. The mocking tests are there as a showcase of how to test the application without real data storage.

In the infrastructure, there are tests again for both Dapper and Entity Framework (EF). These tests are for repositories and queries. There is a total of seventy-four tests in the project.



```
CMS_PV179 (74 tests)
  Dapper.Infrastructure.Tests (10)
    Dapper.Infrastructure.Tests.Query (5)
    Dapper.Infrastructure.Tests.Repository (5)
  EntityFramework.Infrastructure.Tests (10)
    EntityFramework.Infrastructure.Tests.Query (5)
    EntityFramework.Infrastructure.Tests.Repository (5)
  Integration.BusinessLayer.Tests (51)
    Integration.BusinessLayer.Tests.Dapper.FacadeTests (26)
    Integration.BusinessLayer.Tests.EntityFramework.FacadeTests (25)
  Mock.BusinessLayer.Tests (3)
    Mock.BusinessLayer.Tests.FacadeTests (2)
    Mock.BusinessLayer.Tests.QueryObjectTests (1)
```

Figure 3.5: Test explorer output

4. https://www.paloaltonetworks.com/cyberpedia/what-is-a-denial-of-service-attack-dos
5. https://www.w3schools.com/sql/sql_injection.asp
6. https://portswigger.net/web-security/cross-site-scripting

3.5 Used technologies

This section describes all the technologies, techinques and third-party libraries that the project is using.

3.5.1 ORMs

Object-Relational Mapping (ORM) is a programming technique that creates a way to manipulate the data from the database with the use of the object-oriented paradigm. When talking about ORM, most people are referring to a library that implements the Object-Relational Mapping technique, hence the name ORM. Developers write the object code in object-oriented programming (OOP) languages. In the case of the project, the language is C#. The ORM's only purpose is to convert the data between different type systems that are not able to coexist within OOP languages and relational databases [11].

In other words, the ORM library is an ordinary library that encapsulates the code needed to manipulate the data, so there is no need to use SQL anymore. Quite the opposite, developers interact directly with an object in the same language they use.

The project is using two different ORMs, and each of them has unique functionality, usability, and advantages. In terms of performance, Dapper is a better choice. On the other hand, in terms of usability, EF is by far the best.

Dapper

Dapper[7] is a simple object-relational mapper .NET and .NET Core. It is a Micro ORM and in terms of speed and efficiency is as fast as using built in .NET functionality for data reading.[12]

Micro ORM means that it is not so robust and have less functionality, but the great advantage is that it is remarkably agile and effective. The disadvantage of Dapper is writing of SQL code. However, there is no need to worry about mapping.

7. https://dapper-tutorial.net/

The project is using a simplistic library over Dapper for CRUD operations called Dapper.SimpleCRUD[8], which provides extension methods to simplify access and omits the need to use raw SQL queries.

Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, open-source, and cross-platform ORM library that provides effortless access to the database without a single line of SQL query [13].

EF Core is the exact opposite of a micro ORM; it is incredibly robust and has almost unlimited functionality. The functionality is growing and upgrading continuously. The best thing about EF is that it is not required to know the framework. In an application, it sees database tables as collections. So developers can access it as collections via built-in API or LINQ.

3.5.2 xUnit

xUnit.net is a free, open-source testing tool for .NET Core. It provides clean solutions for the most common scenarios in testing. It is also fit to test with other .NET languages such as C#, F#, VB.NET [14].

xUnit is a part of all the test projects as the only testing tool. It has plenty of features, most significant of them is ClassFixture, which come in very handy. It merely creates one test class instance for all the tests inside, meaning the possibility to do more tests on it. For example, the test class contains more tests, and they need to run on the same database context. Configuring the order of the tests is the only thing the application needs.

3.5.3 Automapper

Automapper is a simple library, that copies one object properties to another object. Getting rid of unnecessary and redundant code is achievable with the help of this tool. It brings a lot of custom configurations, where the specification of how it should map the given object is. In the project, the primary usage of Automapper is mapping the business entities to DTOs. In smaller projects, it is not as

8. https://github.com/ericdc1/Dapper.SimpleCRUD

efficient, because there would not be so much code. However, imagine the example of having twenty entities and fifty DTOs. The difference between the amount of code would be enormous. In a smaller project, it is not as efficient, because there would not be so much code.

3.5.4 ASP.NET Web API

ASP.NET Web API is an application programming interface for web servers or web browsers [1]. Communication between API and user is going through HTTP requests. Web API is using endpoints, which in most cases returns JSON or XML result. The endpoints contain almost the same logic as MVC controllers. There are two types of Web API.

Client-Side API, extending the logic for interactive websites that are using JavaScript and AJAX[9]. These web APIs can return any data.

Server-Side are those returning JSON and XML data primarily. Mobile applications use this type of API as a bridge to the database. This type of API, in combination with JavaScript frameworks, can create Single Page Applications (SPA). There are also several types of authorization within the API. The most common is Auth2 or Bearer token. The project is using API endpoints to gather data for the Angular presentation layer.

3.5.5 Angular

Angular is an application framework. It is Type-Script based, which is JavaScript with types and classes. This framework is also open-source and has a massive community. It combines dependency injection, best practices for solving challenges, and declarative templates. Angular is a platform that makes it easy to build applications with the web. Developers can use angular to develop the web, mobile or desktop applications [15].

It is also a significantly powerful tool to create Single-Page Applications (SPA), which are applications that are interacting dynamically. It means that the content is rewriting without the need to refresh or redirect and the page skeleton remains the same. User's browser loads all the necessary code like HTML, JavaScript, and CSS on the first visit

9. https://www.w3schools.com/js/js_ajax_intro.asp

of the application. Thus there is no need to demand it from the server in every request.

3.5.6 DotVVM

DotVVM is an open-source component-based framework. It simplifies the building of web applications. It stands as a visual studio plugin [16].

Advantage of this framework, apart from Angular, is that the learning curve is much smaller because all the code is in C#, and it is very understandable and intuitive. Even more, if the developer is comfortable with MVVM architecture[10], which is a two-way binding architecture commonly used in desktop applications.

In the project, there is a full SPA presentation layer implemented using the DotVVM framework.

3.5.7 Materialize CSS

Materialize[11] is a lightweight CSS framework, which is fully responsive. It takes inspiration in Google's Material Design which is a design language, that combines traditional principles of design along with modernization.

The reason the project is using this framework is that apart from Twitter's Bootstrap[12], is independent of JQuery[13], which is a robust framework, but heavyweight, and not necessary for the project. However, the usage and components are almost the same.

10. https://en.wikipedia.org/wiki/Model-view-viewmodel
11. https://materializecss.com
12. https://getbootstrap.com/
13. https://jquery.com/

4 Conclusion

The main goal of this thesis was to develop a complex web application – content management system.

In the first chapter of the theoretical part, the main differences between the ASP.NET Core and ASP.NET Framework technologies are explained and compared to each other. Afterwards, the second chapter primarily focused on design patterns. The following third chapter contained an analysis of the implementation, architecture and structure of the project. Additionally, it also described used third-party technologies and libraries in the project.

The result of the practical part of this thesis is a web application project split into twenty-two iterations, whereas the first one is starting from scratch, and the last one is a fully functioning content management system. The purpose of the project is to serve as study material for PV179 course, where those iterations resulting in eleven tuples. Each tuple consists of two projects - task and solution. The students should implement what is in the task project in class, and the result of their doing should be the solution project.

Part of the thesis is a catalogue of problems that might occur during the development of web application in a given technology and their potential solutions.

In the future, there might be several possible improvements. One of them is to migrate the project to the newer version of .NET Core, which is .NET Core 3.0 that was released at the end of the year 2019. Moreover, there could be more complex integration testing for the application in terms of the presentation layer. Another one of the improvements could be full implementation of the presentation layer in JavaScript framework or entirely new technology released with the mentioned .NET Core 3.0 - Blazor.

Attachments

To the electronic version of the thesis were attached the following files:

- **Project**: Contains full source code for every iteration of the project.
- **Catalog.pdf**: Is a catalog contaning the most common issues and problems that can occur, when developing the project.

Bibliography

1. FREEMAN, Adam. *Pro ASP.NET Core MVC 2: Develop cloud-ready web applications using Microsoft's latest framework, ASP.NET Core MVC 2*. London (UK): Apress, 2017. ISBN 978-1-4842-3150-0.
2. *Differences between .NET Core and .NET Framework - Medium* [online]. US: A Medium Corporation, 2019 [visited on 2019-10-05]. Available from: `https://medium.com/@mindfire%5C-solutions.usa/difference-between-net-core-and-net-framework-c0588e734b99`.
3. *Middleware - Microsoft* [online]. US: Microsoft, 2019 [visited on 2019-10-05]. Available from: `https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.2`.
4. METZGAR, Dustin. *NET Core in action*. Shelter Island, NY: Manning Publications, 2018. ISBN 978-1617294273.
5. GAMMA, Erich. *Design patterns : elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995. ISBN 978-0201633610.
6. FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, 2003. ISBN 978-0321127426.
7. DAIGNEAU, Robert. *Service design patterns : fundamental design solutions for SOAP/WSDL and restful Web services*. Upper Saddle River, NJ: Addison-Wesley, 2012. ISBN 978-0321544209.
8. *Design Patterns — A quick guide to Facade pattern.* [online]. US: A Medium Corporation, 2019 [visited on 2019-13-05]. Available from: `https://medium.com/@andreaspoyias/design-patterns-a-quick-guide-to-facade-pattern-16e3d2f1bfb6`.
9. *MVC - Codecademy* [online]. US: Codecademy, 2019 [visited on 2019-20-05]. Available from: `https://www.codecademy.com/articles/mvc`.
10. *3-Tier architecture Jinfonet Software, Inc.* [online]. US: Jinfonet Software, Inc., 2019 [visited on 2019-10-05]. Available from: `https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/`.

11. *What is ORM - Techopedia* [online]. US: Techopedia, 2019 [visited on 2019-20-05]. Available from: `https://www.techopedia.com/definition/24200/object-relational-mapping--orm`.

12. *What is Dapper - Dapper* [online]. US: Dapper, 2019 [visited on 2019-21-05]. Available from: `https://dapper-tutorial.net/dapper`.

13. *Entity Framework Core - Microsoft* [online]. US: Microsoft, 2019 [visited on 2019-10-05]. Available from: `https://docs.microsoft.com/en-us/ef/core/`.

14. *What is xUnit - xUnit* [online]. US: xUnit.net, 2019 [visited on 2019-21-05]. Available from: `https://xunit.net/`.

15. *What is Angular? - Angular* [online]. US: Angular, 2019 [visited on 2019-21-05]. Available from: `https://angular.io/docs`.

16. *Component-based MVVM framework - DotVVM* [online]. US: DotVVM, 2019 [visited on 2019-21-05]. Available from: `https://www.dotvvm.com/`.