Masaryk University
Faculty of Informatics

Study materials for the web development in .NET Core

Bachelor's Thesis

Viliam Popróči

Brno, Spring 2019

Masaryk University
Faculty of Informatics

Study materials for the web development in .NET Core

Bachelor's Thesis

Viliam Popróči

Brno, Spring 2019

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

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 me a different perspective on the topic.

I would also like to express my gratitude for the support of my family and friends.

Abstract

The goal of this thesis is to develop a complex web application, which serves as educational material for the PV179 course and describe used design patterns and architecture. 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 and architecture. The last part contains the project structure and the technologies used in it.

Keywords

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

Contents

List of Figures

Introduction

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

That is the reason why I am writing this thesis. The primary goal of it is to implement web application, which will serve as study material for PV179 course in newer technology, ASP.NET Core since the currently implemented application is written in an older technology called ASP.NET framework. It cannot be said that the ASP.NET Framework is outdated, but the time will come, when it will no longer have new releases, meaning that one day it may lose compatibility, even if that is very unlikely. However, considering how many new features the new ASP.NET Core is bringing, it would be a pity, not to take advantage of it.

As was said, the project is a web application implemented in C# ASP.NET Core. It is a content management system to be precise, which consists of twenty-four iterations, wherein the first one we are starting from scratch, and in the last one, we have a fully functioning application with all the features content management system should have, while emphasizing on the correct usage of known design patterns and architectures.

The first chapter of this thesis describes the main differences between the mentioned technologies and comparing them to each other.

The second chapter focuses primarily on all the design patterns and architectures that are used in the project and describing why and when to use them.

The third chapter analyzes the implementation and structure of the project and describes the third-party technologies that are used in the project and why are they helpful or necessary.

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 MVC is a web application development framework from Microsoft that combines the effectiveness and tidiness of model-view-controller (MVC) architecture, ideas, and techniques from agile development, and the best parts of the .NET platform.[1]

The main motivations to use ASP.NET Core in our application is that it is running on .NET Core, which is similar to .NET Framework, but a cross-platform, not Windows dependent version of it. Although Windows is still leading operating system in terms of usage, web applications are more and more being hosted on smaller and simpler containers in the cloud platform. Microsoft extended its reach of .NET by embracing the cross-platform strategy and making it possible to host ASP.NET Core applications in a broader spectrum of hosting environments. The bonus of this is that developers can create their web applications on Linux and macOS. Moreover, .NET Core is fully open source, meaning we can download the source codes of the framework, rewrite it, and compile our modified version of it, which may come in remarkably handy. When we run into system component error, and we can debug it out with the privilege of reading the original programmer comments, or when we want to find out the limits of the component when we are developing something more advanced.

Besides Visual Studio, which is likely 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 our web application.

1.2.1 Installation

When we are installing the .NET Framework, we need to install it as a single package and runtime environment for Windows. However, as was mentioned, .NET Core is cross-platform and needs to be packaged and installed independent of the underlying operating system.[2] All of the NuGet [1] packages and dependencies need to be compiled and included in our .NET Core applications.

1.2.2 Project structure

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

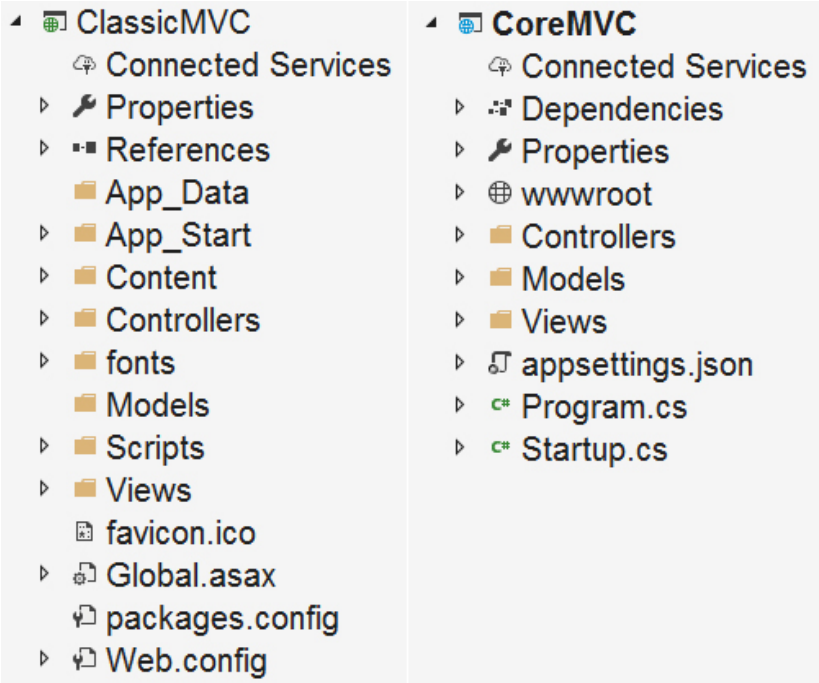

Figure 1.1: Project structure

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

The first thing we can see is that we no longer have References item, but Dependencies instead. The difference between these two may appear as it is only a change of the name since in terms of logic, both serve the same purpose. However, as mentioned above, the References are referring to installed package, whereas Dependencies refers to packages inside our application and they will appear in our bin/debug folder after we build it.

In our full .NET Framework applications, we have several folders like Scripts, App_Data, App_Start, Fonts, Content, and files like favicon.ico, unlike ASP.NET Core, which only has one folder named the wwwroot that contains all of the website static files. It is only a cosmetic change to clean up the main folder. The rest of the folders - Models, Views, Controllers - are the same.

The application configuration in ASP.NET Core is placed in the JSON file named appsettings. We may consider this as our web.config file, which we can find in our .NET Framework application. However, the significant difference is that web.config is structured in XML and is difficult to read and write, whereas in ASP.NET Core application settings are structured in JSON and are very simplified, which makes them rather easy to write and read. On the other hand, we did not get completely rid of web.config, because if we would want to host our application on Internet Information Services (IIS), the web.config file is still generated and attached to the resulting DLL.

The code, in most parts, is the same as far as for controllers and views goes, although there are slight differences. We can now use tag helpers in views, which in terms of logic are doing the same thing, but looks much tidier in terms of semantics. On the other hand, we are still able to use the same razor syntax and same helper as before, if we are more comfortable with it. However, there are many changes in the code of configuration, meaning Startup.cs and Global.asax.

1.2.3 Configuration

Whole ASP.NET Core and ASP.NET is running on a technology called middleware. Middleware is software that is assembled into an app pipeline to handle requests and responses.[3] We can imagine it as a queue of people handing some package to the one behind when the package arrives at the last person, he opens it and sends it back to the

front. Now imagine our HTTP request is our package as we can see on 1.2.

Figure 1.2: Middleware illustration

In other words, we are configuring our applications by adding, creating, and ordering this middleware in our Startup.cs. However, the most significant change in configuration is that in ASP.NET Core, we now have native dependency injection, so we no longer need third-party libraries for resolving our dependencies. Moreover, injecting is very intuitive and clean.

2 Design Patterns

This chapter focuses primarily on design patterns we are using in the project.

2.1 Three-Tier architecture

A Three-tier architecture is a type of software architecture which is composed of three „tiers“ or „layers“ of logical computing.[4] We can see the simplistic version of the architecture in 2.1.

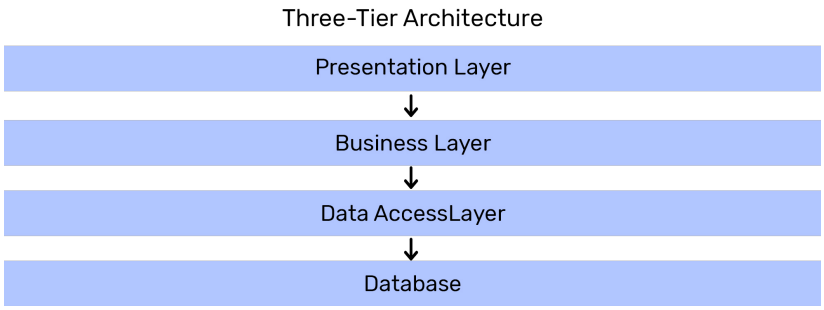

Figure 2.1: Simplified Three-Tier architecture

The real question is, what do those layers mean, and what is the motivation of using this architecture? The main reason is to separate the application into three tiers, where every layer has its single purpose, and each is dependent strictly to the one below.

The data access layer (DAL), as the name suggests, serves as our communication with the database. In most cases, we have our SQL procedures or Object-relational mapping (ORM) tool - see 3.4.1, which maps the tables to the classes, so we can have better control over our entities effortlessly.

The business layer (BL) serves as communication with the DAL, and inside, we have our business logic, meaning the operations over our entities, their interaction with each other and calculations.

In the top, we have our presentation layer (PL), where we mostly store our user interface (UI) components, meaning javascript, HTML,

CSS. UI is often accessible from the web browser and displays the content for the end user, which came from the BL.

The main benefits of Three-Tier architecture are the speed of development, scalability, performance, and availability. It is pretty easy to split the work between more developers, meaning everyone can focus on their work, and it is beneficial because we can change some code in one layer and it will have almost zero impact on the other one. Meaning 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, we would have a server for each layer, and that is where scalability comes in because, in a situation where there are many requests on the BL, we would only scale that server without the need to touch other. That brings another advantage because we can manage our resources so much better.

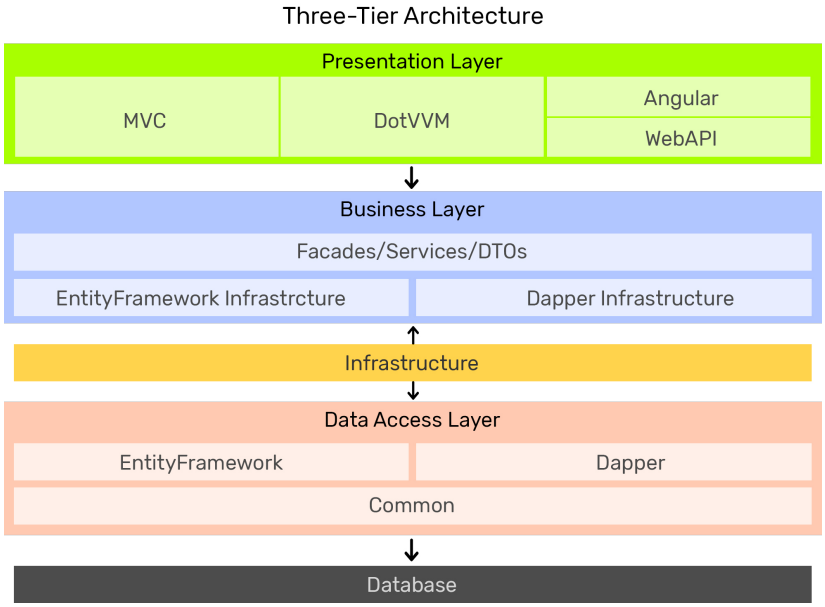

Figure 2.2: Three-Tier architecture used in the project

On the other hand, it is not always worth it, to go for Three-Tier architecture because it can be very time-consuming to prepare it. In a situation, where we would want to build only a small application,

it would take more time to make the architecture, then to build the application itself.

In the 2.2, we can see how the architecture look in the project.

2.2 Unit of Work

Unit of work (UOW) is the level of abstraction, which has two jobs to do. Maintain the information we want to pass to the database, and pass information to the database. We can understand it as a transaction, so when something goes wrong in the middle of the process, the database rolls back to the state before the transaction.

To understand it correctly, we can split UOW to single words. Work in terms of software application is nothing other than just inserting, updating, and deleting data from the database. Whereas the unit stands for a single unit, which has to keep track of what is going on in a single transaction, meaning that we can do multiple „works“ within it.

```
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 2.3: Unit of work interface

In the project, we have the 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 pretty straightforward, and we do not have to worry about some unwanted data in the database. In 2.3, we can see how the interface looks. At the start of the transaction,

we call the provider's Create method, and at the end of the transaction, we are calling UOW's Commit.

2.3 Repository

The repository is another level of abstraction placed in the project infrastructure - see 2.2. The real purpose of the repository is to create communication between the data access layer and business layer, meaning that we get the entities from the database and map them to our business entities, with or without ORM. In most cases, 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 easiest one is to have one repository per business entity. However, this is highly unpractical, because we would have too much redundant code. The second and probably the tidiest way is to make a generic repository, which is the way we are going in the project. In the project, we are using the combination of the Repository and Query Object pattern - see the following section. Most developers use the repository pattern without even realizing it because Entity Framework has a built-in implementation of it. However, since we are using more than one ORM because we want to show how easy it is to swap the data access layer, without changing the API. For instance, we would use several databases, but the repository implementation would still look the same. The point is that the repository will provide a clean API from which BL easily read and write to it, without worrying where the data is, in fact going. Here is how our repository looks like; the methods are self-explanatory:

```
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 2.4: Repository interface

2.4 Query object

The query object is placed in infrastructure too, and it has a similar purpose as the Repository. However, the difference is that we could use the only Repository, but we would have one for each entity, which, as was mentioned, is indeed not the way we want to go. What is the difference, then? The Repository only serves for CRUD operations on a single entity, whereas Query Object is only for retrieving, but multiple entities. In most cases, we want our data to run through some filter. For example, we have a list of images, and we want to pick only the first ten rows ordered by date of creation. That is precisely the use case of the query object.

The thing here is that the implementation might be much trickier if we want to do it as effectively as possible. We could implement the query object throughout Expression trees and reflexion, but that will not give us the performance we want. In the project, we are connecting strings of SQL into resulting SQL query, which we are evaluating on the database, which might be the fastest way to do it. This is how the query object in the project looks like:

```
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 2.5: Query interface

The IPredicate is an interface for our predicate structure, there are two types of predicates Simple predicate and Composite predicate. Simple predicate is just one condition with value comparing operator and Composite predicate is an array of IPredicates with logical operator.

The QueryResult is our class that represent the final result of our Query:

```
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 2.6: QueryResult class

2.5 Service

Service is another level of abstraction above the query objects and repository, except that mapping, the database entities to Data Transfer Objects (DTOs) (following section) is happening here. This layer is uniting both of them together. We have single service for every entity, this may sound like that this is much of unnecessary code, but the fact is that we can implement it very briefly with the generics, for example, CRUD operations. The reason we have services is that here we are not only doing CRUD services, but also some calculations and other operations. For example, we are computing the hash and salt for passwords, resizing images, or shrinking documents. It is also a layer where we are not dependent on the data access layer because repository and query objects handle all the storing logic.

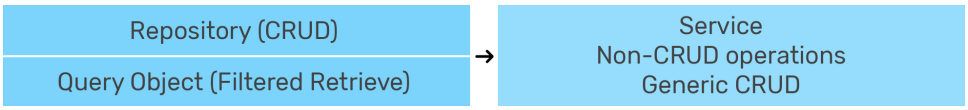

Figure 2.7: Service diagram

2.6 Facade

The facade pattern is a software-design pattern commonly used in object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code.[5]

In the project, we are using facades to combine multiple services.

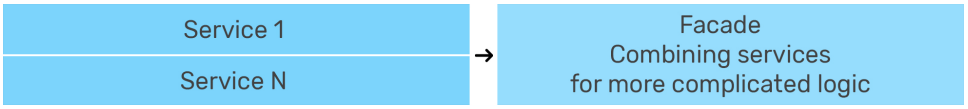

Figure 2.8: Facade diagram

The services are providing us with needed functionality, and by combing them together, we can make more complex solutions in this layer. That may come very handy because we can make operations we would normally do in controllers, which will now contain almost zero application logic. Moreover, in our Presentation layer, we no longer need any dependencies needed in the Business layer, if we would skip facades and only used services, we would at least needed dependency for our mapping library.

2.7 Data Transfer Object

Data Transfer Object (DTO) is the object that carries data between layers. The motivation to use it to reduce the size of transferred objects, and sometimes we do not want to share all the information that comes from the database, for example, we never want to retrieve user password to the presentation layer, because it may cause some security breaches. Instead, we just want to pass the absolutely needed information. DTOs does not have any behavior, meaning they only serve one purpose, which is to transfer data. However, they might contain some serialization logic because REST service often uses them as a returning type. We can easily imagine that on example, where we have a File object in the database, and we would want to send it throughout REST into some mobile device. It would be very slow and ineffective.

2.8 Model-View-Controller

MVC is short for Model, View, and Controller. MVC is a popular way of organizing code. The big idea behind MVC is that each section of the code has a purpose, and those purposes are different. Some of the code holds the data of the application, some of the code makes the application look likable, and some of the code controls how the application is functioning.[6]

In most cases, Model represents a real-world thing. We can say that the DTO we retrieve from the facade is our model.

The controller contains all the logic, which we want to invoke on the model, for example, in the project, we retrieve the model from the facade and then, we populate it to the view.

The view is the eye candy, meaning that all our graphics and what user sees, in reality, goes to the view (HTML, CSS).

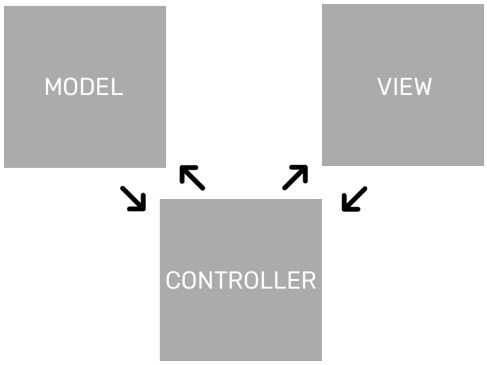

Figure 2.9: MVC flow

In 2.9, we can see how the flow works. Firstly in the controller, we get our model, and then we populate the view with it. Then we register user action in view, and we are returning to the controller, which updates the model and returning it once again.

3 Project

This chapter contains the details about the project implementation and structure. It also contains information about the technologies and libraries that we are using in the project.

First of all, what is the project about? It is a simple content management system (CMS) [1], which has several functions. We can create and manage our posts, pages, and articles, comment on them, assign them to categories. Control our images, videos, documents, and other files. Moreover, we can register, login and manage our content in the administrator section.

3.1 Class Diagram

Although the class diagram[2] in the project is rather simplistic, we can demonstrate the most common use cases on it.

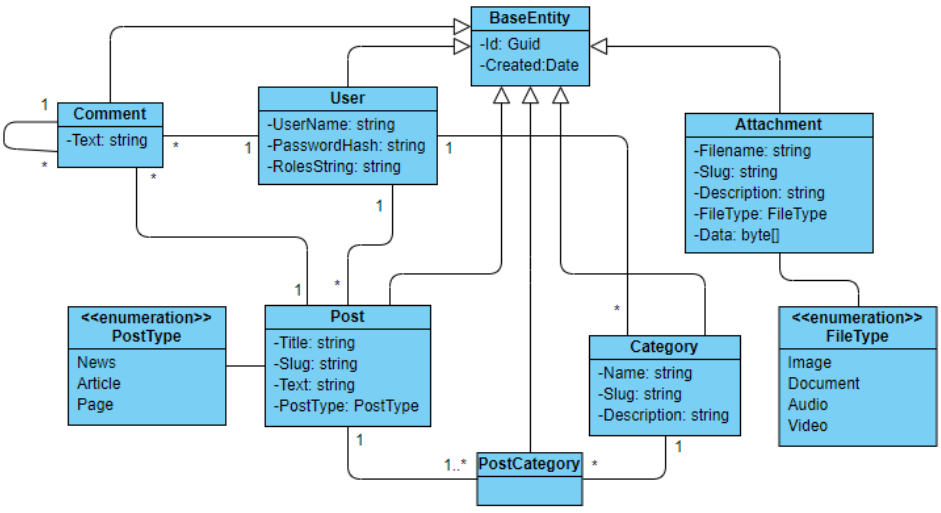

Figure 3.1: Class diagram for the project

1. https://en.wikipedia.org/wiki/Content_management_system
2. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

3.2 Requirements

The only requirement for this project is to have .NET Core 2.2 SDK[3] and Visual Studio or Visual Studio Code IDE[4].

3.3 Structure

3.3.1 Zip file structure

The zip file contains twenty-four iterations of the project; they are making tuples called "labX/task" and "labX/finished". Where task folders contain unfinished project iteration and todo list with information for instructors of the course, as name finished suggest, the finished folder includes the complete version of given task iteration.

3.3.2 Last solution iteration

The final project is rather complicated, so in this section, we will go through the entire solution. In 3.2, we can see the entire solution structure and all the sub-projects contained in it.

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

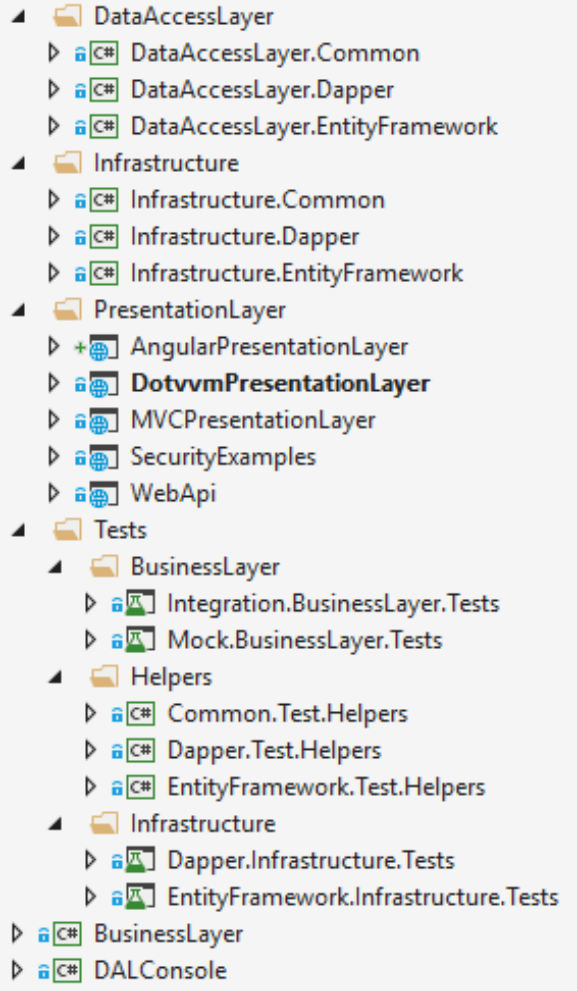

Figure 3.2: Entire solution structure

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 our data called DbSets.

Infrastructure folder

Another similarly ordered and named projects. The **Common** contains mainly interfaces for our infrastructure, whereas the rest of them focuses on the real implementation. The infrastructure contains the implementation of Query Object, Repository, and Predicates. The Query object and Repository was mentioned and showed before.

BusinessLayer project

In this project, all the application logic is, meaning all the services, DTOs, Query Objects, Facades, and mapping configuration. All of these was mentioned before, except mapping configuration, which is a file where we specify how our database entities should be mapping into DTOs - see 3.4.3.

PresentationLayer folder

In this folder, we have five projects; the first one is **AngularPresentationLayer**. This project serves only as an example introduction to javascript framework called Angular - 3.4.5.

The second project is **DotvvmPresentationLayer**, which contains fully functioning PL on a framework called DotVVM - 3.4.6.

MVCPresentationLayer, as the name suggests, is classic complete PL build on classic MVC and Razor Pages. This project is the primary 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)[5] prevention, SQL Injection[6], or Cross-Site scripting (XSS)[7].

The last project is our **WebApi**, which serves as our REST application, and it is necessary to have it running when we want the Angular project to run correctly.

---

5. https://www.paloaltonetworks.com/cyberpedia/what-is-a-denial-of-service-attack-dos
6. https://www.w3schools.com/sql/sql_injection.asp
7. https://sk.wikipedia.org/wiki/Cross-site_scripting

Tests Folder

In the folder, we have several other folders, on the lowest level, there are three class libraries with Helpers for the actual tests, containing some overrides, data seeds, and configuration.

The other two folders are pretty self-explanatory. However, the tests in BusinessLayer the Integration tests project contains tests for the facade with both infrastructures, meaning Dapper and Entity Framework. Mocking tests are there as a showcase, how we can test without the database.

In the infrastructure, there are tests again for both Dapper and Entity Framework, testing repositories and queries. We have seventy-four tests in total in the project.

Figure 3.3: Test explorer output

3.4 Used technologies

In this section we will discuss all the technologies and third-party libraries that the project is using.

3.4.1 ORMs

Object-Relational Mapping (ORM) is a programming technique that lets you retrieve and manipulate data from/to the database using an 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. Object code is written in object-oriented programming languages. In the case of the project C#. ORM converts data between type systems that are unable to coexist within relational databases and OOP languages.[7]

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, we are interacting directly with an object in the same language we are using.

In the project, we are using two different ORMs, and each of them has unique functionality, usability, and advantages. In terms of performance, Dapper is a better choice.

Dapper

Dapper[8] is a simple object mapper for .NET and own the title of King of Micro ORM in terms of speed and is virtually as fast as using a raw ADO.NET data reader.[8]

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 thing about dapper is that we are writing SQL code. However, we do not have to worry about mapping.

In the project, we are also using the library for CRUD operations with dapper called Dapper.SimpleCRUD[9], meaning that we do not have to worry about the SQL, even in this case.

8. https://dapper-tutorial.net/
9. https://github.com/ericdc1/Dapper.SimpleCRUD

Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, open source, and cross-platform version of the popular Entity Framework data access technology.[9]

EF Core is the exact opposite of a micro ORM; it is incredibly robust and has almost unlimited functionality. In fact, we do not even need to know the framework; we only need to know how to write LINQ[10].

3.4.2 XUnit

xUnit.net is a free, open source, community-focused unit testing tool for the .NET Framework. Written by the original inventor of NUnit v2, xUnit.net is the latest technology for unit testing C#, F#, VB.NET, and other .NET languages.[10]

In all test projects, we are using xUnit as the only testing tool. It has plenty of features, most significant of them is ClassFixture, which come in very handy. It simply creates one test class instance for all the tests inside, meaning the possibility to do more tests on, for example, single database context, where we would just have to choose the order of the tests to be run.

3.4.3 Automapper

Automapper[11] is a simple library, that copies one object properties to another object. With the help of it, we can get rid of unnecessary and annoying to write code. It brings a lot of custom configurations, where we can specify how it should map the given object. In the project, the primary usage of Automapper is mapping the database entities to DTOs. In smaller project it is not as efficient, because there would not be so much code. But if we would have, for example, 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. However, if we would have, for example,

10. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/getting-started-with-linq
11. https://automapper.org/

twenty entities and fifty DTOs, the difference between the amount of the written code would be enormous.

3.4.4 Web API

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. We have two types of Web API.

Client-Side API, extending the logic for interactive websites that are using javascript and AJAX[12]. These web APIs can return any data.

Server-Side are those returning JSON and XML data primarily and are mostly used in mobile applications 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. In the project we are using API endpoints to gather data for the Angular presentation layer.

3.4.5 Angular

Angular is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop.[11]

It is also a significantly powerful tool to create Single-Page Applications (SPA), which are applications that are interacting dynamically, meaning that the content is rewriting without the need to refresh or redirect and the page skeleton is still the same. After the first load of the page, all the necessary code like HTML, javascript, and CSS is loaded into the client browser. Thus there is no need to request server for it in every request.

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

3.4.6 DotVVM

Thus there is no need to request server for it in every request. DotVVM is a component-based framework that simplifies the building of enterprise web applications. It is open source, integrates with ASP.NET Core and comes with a neat Visual Studio integration.[12]

Advantage of this framework apart from Angular, is that the learning curve is much smaller because all the code is written in C#, and it is very understandable and intuitive. Even more, if the developer is comfortable with MVVM architecture[13], which is a two-way binding architecture used commonly in desktop applications.

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

3.4.7 Materialize CSS

Materialize[14] is a fully responsive CSS framework based on Google's Material Design, which is a design language combining traditional principles of successful design along with modernization and technology.

The reason the project is using this framework is that apart from Twitter's Bootstrap[15], is independent of JQuery[16], which is a robust framework, but heavyweight, and not necessary for the project. However, the usage and components are almost the same.

13. https://en.wikipedia.org/wiki/Model-view-viewmodel
14. https://materializecss.com
15. https://getbootstrap.com/
16. https://jquery.com/

4 Conclusion

The result of this thesis is a web application project split into twenty-four iterations, whereas the first one is starting from scratch, and the last one is a fully functioning content management system.

The purpose of this project is to serve as study material for PV179 course, where those iterations are combined into tuples projects task and solution for that task. The students should implement what is in the task project in class, and the result should be the solution project.

In the future, there might be several possible improvements. One of them is to migrate the project to the newer version of .NET Core, .NET Core 3.0 to be exact, which will be released later this year. Moreover, there could be more integration tests for the application in terms of the presentation layer. Another one of the improvements could full implementation of presentation layer in javascript framework.

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*. 7th ed. 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. *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/.
5. *Facade pattern - Wikipedia* [online]. US: Wikipedia, 2019 [visited on 2019-13-05]. Available from: https://en.wikipedia.org/wiki/Facade%5C_pattern.
6. *MVC - Codecademy* [online]. US: Codecademy, 2019 [visited on 2019-20-05]. Available from: https://www.codecademy.com/articles/mvc.
7. *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.
8. *What is Dapper - Dapper* [online]. US: Dapper, 2019 [visited on 2019-21-05]. Available from: https://dapper-tutorial.net/dapper.
9. *Entity Framework Core - Microsoft* [online]. US: Microsoft, 2019 [visited on 2019-10-05]. Available from: https://docs.microsoft.com/en-us/ef/core/.

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

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

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