

University College of Northern Denmark

IT-programme

AP Degree in Computer Science

Dmaj0919

SOLVR.ONLINE

TECHNICAL DOCUMENTATION

Repository:

https://github.com/sebaholesz/semester-3-project

Maros Cuninka, Sebastian Holesz, Samuel Horacek, Martin Hotka, Ioan-Sebastian Voinea

Aalborg, 21. 12. 2020



University College of Northern Denmark

IT-programme

AP Degree in Computer Science

**Class:** Dmaj0919

**Title:** SOLVR.ONLINE - TECHNICAL DOCUMENTATION

**Abstract:**

**Project participants:**

Maros Cuninka

Sebastian Holesz

Samuel Horacek

Martin Hotka

Ioan-Sebastian Voinea

**Supervisor**:

Nadeem Iftikhar

**Submission date:** 21. 12. 2020

Contents

[**INTRODUCTION** 3](#_Toc59308107)

[**1.** **Domain model** 3](#_Toc59308108)

[**2.** **Database design** 3](#_Toc59308109)

[**2.1.** **Relational Model** 3](#_Toc59308110)

[**2.2.** **Object-relational mapping** 4](#_Toc59308111)

[**2.3.** **Database first & Coding first** 4](#_Toc59308112)

[**3.** **Architecture design** 5](#_Toc59308113)

[**3.1.** **Communication protocols** 5](#_Toc59308114)

[**3.2.** **Clients** 6](#_Toc59308115)

[**3.2.1.** **Web Client** 6](#_Toc59308116)

[**3.2.2.** **Desktop client** 9](#_Toc59308117)

[**4.** **Security** 12](#_Toc59308118)

[**4.1.** **SQL Injection** 12](#_Toc59308119)

[**4.2.** **Cross Site Scripting** 12](#_Toc59308120)

[**4.3.** **Cross Site Request Forgery** 13](#_Toc59308121)

[**4.4.** **Brute Force Attack** 14](#_Toc59308122)

[**4.5.** **Distributed Denial Of Service Attack** 14](#_Toc59308123)

[**4.6.** **Man In The Middle Attack** 14](#_Toc59308124)

[**4.7.** **Over Posting attack** 15](#_Toc59308125)

[**5.** **Concurrency** 15](#_Toc59308126)

[**5.1.** **Optimistic vs. pessimistic** 15](#_Toc59308127)

[**5.2.** **Transactions** 16](#_Toc59308128)

[**5.3.** **Our concurrency** 17](#_Toc59308129)

[**6.** **Testing** 23](#_Toc59308130)

[**6.1.** **Unit testing** 23](#_Toc59308131)

[**7.** **Code standards** 24](#_Toc59308132)

[**8.** **Implementation** 25](#_Toc59308133)

[**8.1.** **Pagination** 25](#_Toc59308134)

[**CONCLUSION** 26](#_Toc59308135)

[**REFERENCES** 26](#_Toc59308136)

[**APPENDIX** 26](#_Toc59308137)

[Appendix A – Domain Model 27](#_Toc59308138)

[Appendix B – Relational Model 28](#_Toc59308139)

# **INTRODUCTION**

Text…

# **Domain model**

Even though agile methodologies suggest, to avoid outdated artifacts, that no or only minimal documentation should be done, the development team decided to create an initial domain model of the system. This artifact was created to put the team on the same page, but also as a starting point for the development process. We believe that the domain model is important in the process of understanding a system, especially one that we had to work with for 5 weeks. It serves as a skeleton on which the system is built on. The domain model also stood as a base for the creation of the relational model. Initially, we wrote down all conceptual classes we considered as crucial for our system, then we connected them based on their relationship and finally we have added the multiplicity (Appendix A).

We also used the domain model to support the clarification of our first core user stories. For example, during sprint 1, the core user stories were “Post an assignment” and “Post a solution”. In this case we looked at the domain model and understood that one user can both be an assignment poster and a solution poster, but at the same time, they cannot solve their own assignment.

# **Database design**

## **Relational Model**

Second step we focused on after domain model was mapping it into the Relational Model. At the beginning of planning, we agreed that we will not be doing comprehensive documentation and diagrams for the project, only if it is necessary. And we felt, that having a relational model will help us setting up the database.

While designing a relational model, we were following the Normal Forms by Edgar F. Codd. For example, we applied 1st normal form when we have split the name for User into two columns – firstName and lastName to achieve atomicity. Also, we applied Id column as a primary key of most relations, so all other values depend on the whole key – 2nd normal form.

You can check our Relation Model in Appendix B. This was a first version of it, which includes all tables, even those we did not have time to implement. During process it was also edited due to implementing Identity for Authentication & Authorization, which replaced our User relation.

## **Object-relational mapping**

Object-relational mapping (ORM) is a programming technique in which a metadata descriptor is used to connect object code to a relational database. In other words, the ORM helps to resolve the mismatch between object code and relational database. There are advantages of using ORM:

* Development and maintenance costs are lowered because the conversion of object-to-table and table-to-object is automated
* Less code compared to embedded SQL

Developers can use ORM with a lot of different languages. For C#, one of the most used is Dapper (King of Micro ORM). [1]

Dapper is an object mapper for .NET and is responsible for mapping between programming language and database. It works by extending IDbConnection class by adding extension methods to query the database. These queries are parameterized, which also prevent from SQL injection attack (more in security chapter). The main advantages of using dapper are its performance, it is lightweight and is working with any database. These are the reason why we also chose dapper as our ORM.

## **Database first & Coding first**

When creating a system which is also connected with the database, developers can choose between 2 approaches (at least) – code first and database first.

Code first means that we firstly create entity classes with all their required fields. From this the entity framework creates database with tables according the definition we set in those classes. In simpler words, the database is created when a programmer runs the code

In Database first approach, developers firstly create database with tables, which are used to create entity data model. The main advantage of this approach is mapping and creating keys and relationships between relations. Another advantage can be graphical user interface while creating tables, e. g. in MSSQL Management Studio.

In our project we used both approaches. Since we were using dapper, which only map an existing database, we had to use database first approach, for getting data about Assignment and Solutions from database. By implementing identity for authentication & authorization through individual user accounts provided by MVC we used entity framework. We created tables firstly in code and then by using migration we updated the database. Same logic was applied for inserting new columns into database (firstName, lastName, profilePicture for User).

# **Architecture design**

## **Communication protocols**

Protocols are sets of rules that must be abided when two devices communicate between each other. They are required, if would not be for their specific format, machines could not communicate. They are incredibly useful because they can authenticate and correct messages or detect errors. With some protocols - for example, the HTTPS (Hypertext transfer Protocol Secure), they serve the purpose of securing systems.

There are several protocols we directly or indirectly used in our project. The TCP (Transmission control Protocol), IP (Internet Protocol) & forementioned HTTPS are all protocols at the core of the whole internet and it would be extremely hard, even impossible to make a modern distributed system without them.

TCP is one of the most popular protocols, which is used for communicating across a network. It takes the senders message, divides it into smaller parts (packets) and sends them across the network. The receiver then reassembles the packets together and gets the message. It also makes sure that the receiver got all the packets, in the same sequence, without any duplications or damages.

Internet Protocol is usually used together with TCP. Essentially, it is an addressing protocol, for determining a good route based on the IP address the individual packets carry. There are two versions of the IP protocol - IPv4 and IPv6.

HTTPS is used for communication on the Internet as we, its users, know it. One is using the client browser and the other is sending the data from the Web server. The data is, as the name suggests, in hypertext format. This part is the same as for HTTP, but the advantage of HTTPS is that it is more secure. It sends the data in an encrypted format, so anyone in the network does not see in plain text your HTTP requests/responses. HTTPS as well as other network protocols has a specific port on which is listens to responses from the network.

We could have also easily come across the FTP (File Transport Protocol), which is used to send files between a client and web server (or generally between machines), or the SMTP (Simple mail transport Protocol) if we had implemented email-confirmation for the users of our system.

## **Clients**

## **Web Client**

**Single Page Application and Multi-Page Application**

In order to develop a web client, there are multiple choices regarding patterns and architectures that have to be taken into consideration. There are two main approaches, each with its advantages and disadvantages.

The first one is the Single Page Application (SPA), which is a type of application that works inside the browser and does not require page reloading, as it is built into one page, and loads all the necessary content using JavaScript. The biggest advantage of SPA is responsiveness. It is fast because most of the resources, like HTML, CSS, and Scripts, are only loaded once, at the beginning of the application lifespan. The only information transmitted back and forth is data.

On the other hand, the second approach is a Multi-Page Application (MPA), it is a more traditional way of web development. Every change made on the application, requests a new page from the server, thus, making it slower than the SPA. The advantage of MPA being an easier search-engine optimization.

Due to the constraints on our project, we decided to use MVC Multi-Page Application.

**MVC**

MVC is the most used Multi-Page Application pattern. It stands for Model View Controller, and it is used to achieve separation of concerns by decoupling the user-interface, the data, and the application logic. With this pattern, the requests are routed to a controller, which communicates with the Web API, sending or retrieving data, that is then dynamically rendered using the models and the views.

The model is a set of classes that have the role of describing the data the application works with. Any code manipulating the data and any business logic should also be contained in the model.

The view is responsible for presenting the content through the user interface. The Razor view engine is used for the views, to dynamically generate web content on the server. The view should only contain logic related to presenting content.

The controller is the one that handles the user interactions and the communication from the user, it also manages the relationship between the Model and the View.

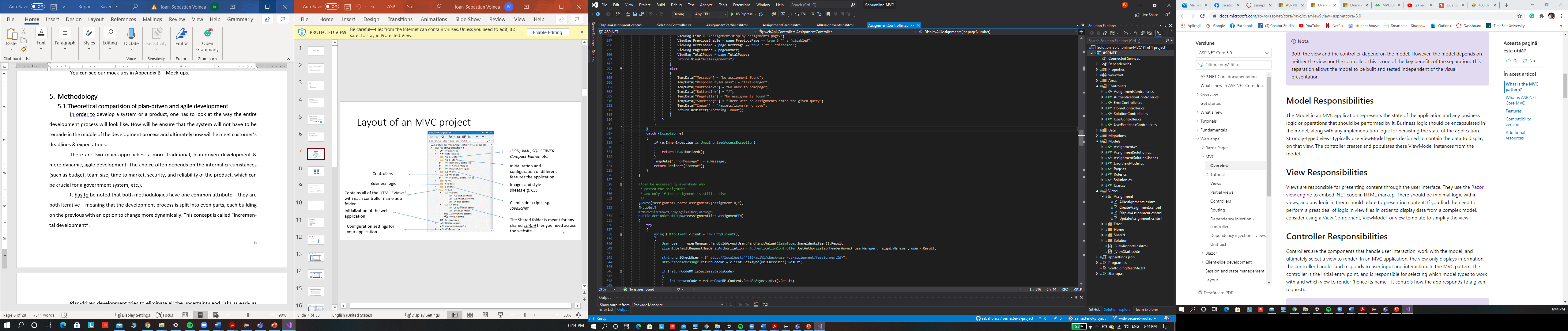
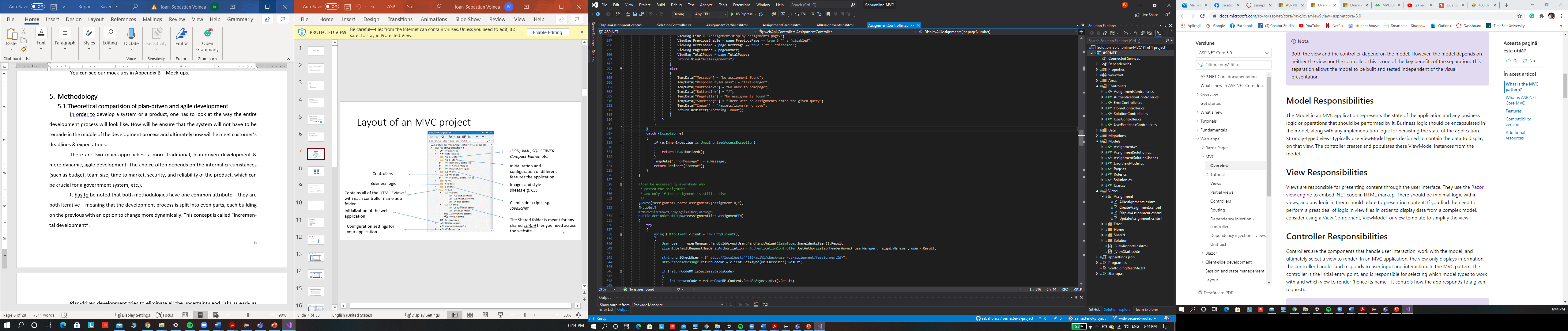


Figure 1 MVC project architecture

**Implementation**

As you can see in the picture below (Figure XXX), when a user wants to create a new assignment, the button “Post Assignment” redirects him to the “/assignment/create-assignment” route.

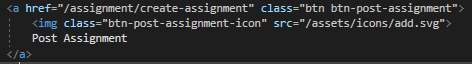


Figure 2 Button Post Assignment

That means an HttpGet request is made to load the new page. When the method is called, before returning the new view, the controller must get all the information needed and pass it to the view.

In the controller below, you can see how the “Create Assignment” method makes HTTP requests to the Web API in order to get all the academic levels and subjects, but also the credits that the user has. After that, we use a ViewBag to dynamically pass the data to the view.

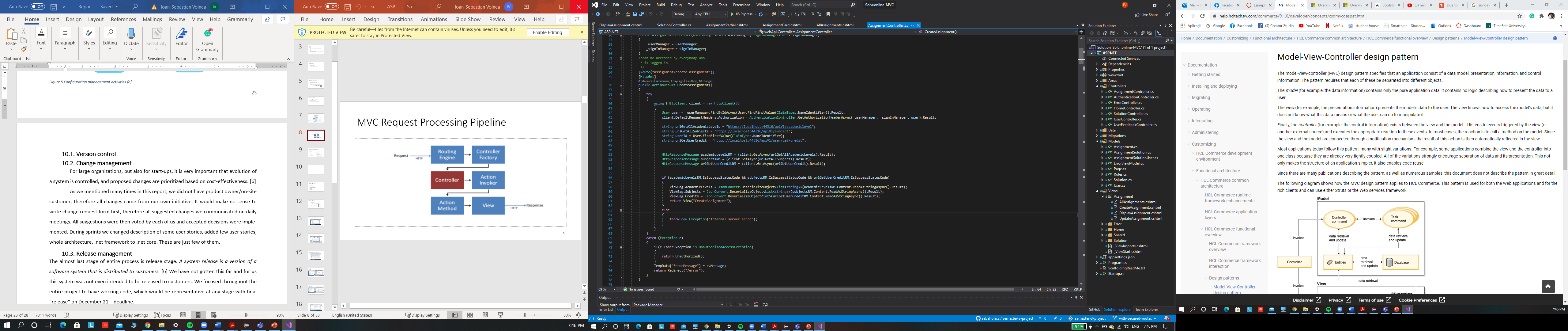


Figure 3 CreateAssignment method in Assignment controller

In the “CreateAssignment” view, we then get the ViewBag object with all the data inside. We use a foreach loop to iterate through all the subjects and academic levels and display them as options on the page.

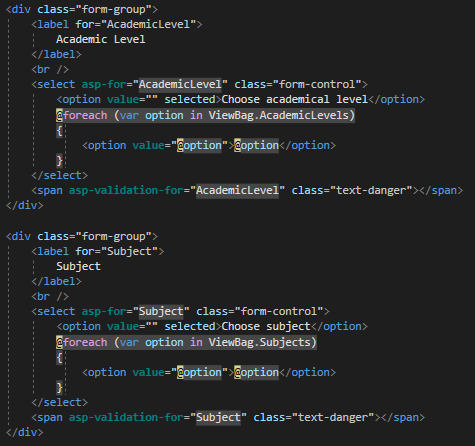


Figure 4 Create assignment view

To keep the view clean, with as little logic as possible, and the code overall DRY, we use the ASP.NET validation. In the assignment model class, we specify what fields are required and the length or range where needed. We then use the “asp-validation-for” to create span messages when the requirements are not met. The ASP.NET validation is also a good way of keeping the application responsive.

**Bootstrap**

Bootstrap is an open-source CSS framework for building web applications. We have used bootstrap in our project because it is easy to use and it saved us a lot of time by using the default Bootstrap classes to style some parts of our application. Due to Bootstrap’s grid system, our web client is also easily resizable.

**Layouts**

To provide the user with a consistent experience throughout their usage of the application, we decided to use the layout feature provided by ASP.NET. For each view, we specify which layout should be rendered at the top of the view by using the Layout property.

For example, for almost all our pages we use the “\_LayoutWithUserSidebar” which includes the navigation bar at the top of the page, and the user sidebar on the left side.

**Partial Views**

In our web application, to avoid code duplication, we use partial views, A partial view is a reusable portion of a web page, it contains HTML code, and it can be used in one or more views or layouts.

One of the most important partial views in our application is the “AssignmentCard”, this view represents a card that contains basic information about an assignment, it is used when displaying a list of multiple assignments. Besides the assignment card, we also use a partial view when displaying complete information about an assignment.

## **Desktop client**

For the desktop application we have decided to use Windows Presentation Foundation (WPF). In this section we will explain our choice of this option and we will also compare it to Windows forms. Let us start with the WPF first.

WPF and Windows forms are both graphical user interfaces used in .Net. The key difference is that WPF is using XAML as markup language, so the programmers can work parallelly with designers. Windows forms are older concept for developing the desktop applications, but on the other hand are easier to use and the learning curve is less. WPF provide better scalability as they are not pixel based instead of Windows forms. Last difference to mention is performance, whereas in WPF are things achieved at faster rate compare to Windows forms.

For our project, we were working with one main window which contained frame with various pages:

* Login page,
* Home page (for all assignments, solutions, and users),
* Update page for assignment.

Working with pages, instead of creating and closing windows, was more convenient and faster. Each page was designed and created in XAML, where we divided it into columns and rows. Then we chose which component we want to use (button, textbox, textblock…) and placed it into specific row and column. With this we achieved good scalability as well.

The main focus with the WPF was not the design itself, but the connection with API. For this we created ApiCalls folder with classes for each model. Those were communicating with the API through the HttpClient and provided us with the result it has received. Like this we could get all assignments or users from the database, update them, or do other operations. One struggle we had, was to implement authorization. As we mentioned before, this desktop application was supposed to be used only by moderators or admins. We dealt with it by creating a bearer token when a user pressed login button, sending his username and password to API where it was checked in the database (Figure XXX).

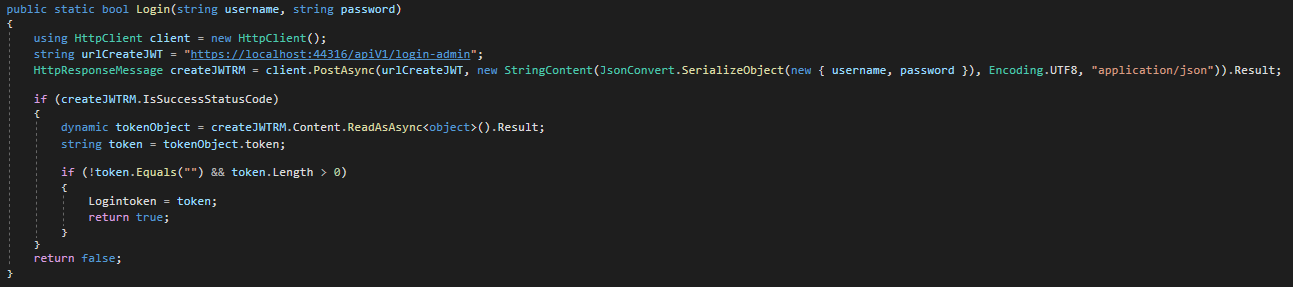


Figure 5 Method for creating bearer token

To achieve higher security, all API calls moderator could use, were checking, if the user who is trying to call them is really moderator. Therefore, for the HttpClient we set the Authorization as request header with his bearer token (Figure XXX).



Figure 6 Communication with API

To display information about assignments and users we used DataGrid component. We used one DataGrid for both to achieve better readability and maintainability, the data just were overridden. Same goes with some buttons and text blocks. Thanks to using WPF, we could also apply different styles for components – rounded corners for buttons, background color and hover color (Figure XXX).

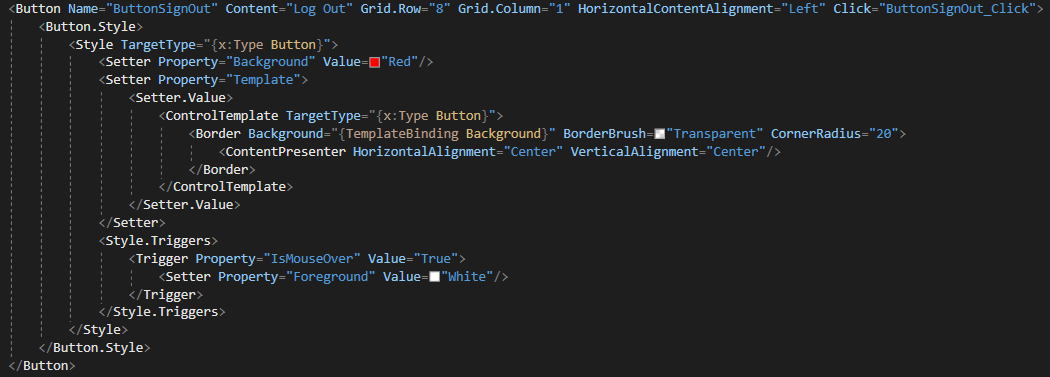


Figure 7 Applying styles for sign out button

# **Security**

## **SQL Injection**

SQL Injection is a common and well spread type of attack, it capitalizes on badly sanitized user inputs. The attacker is trying to exploit this vulnerability by inserting his own SQL commands into a search bar, login field or any other input field, accessing database tables to which he would normally not have permission. Aftermath of this vulnerability can result into severe damage to the company by the leak of sensitive information or lost data. How to simply avoid this is to use parametrized arguments in the SQL queries of your Data Access Layer. In our case we used Dapper which is a lightweight ORM that makes it convenient to do fully parametrized queries without using any input concatenation.

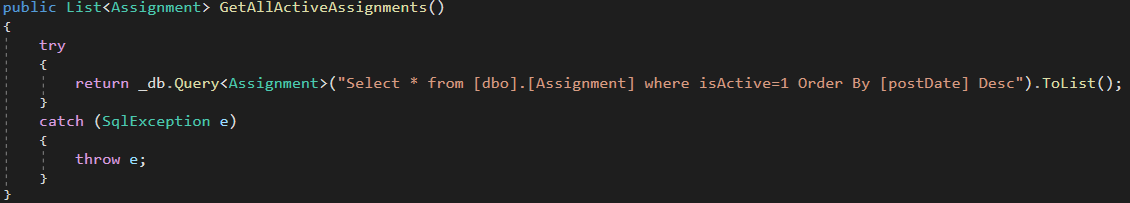
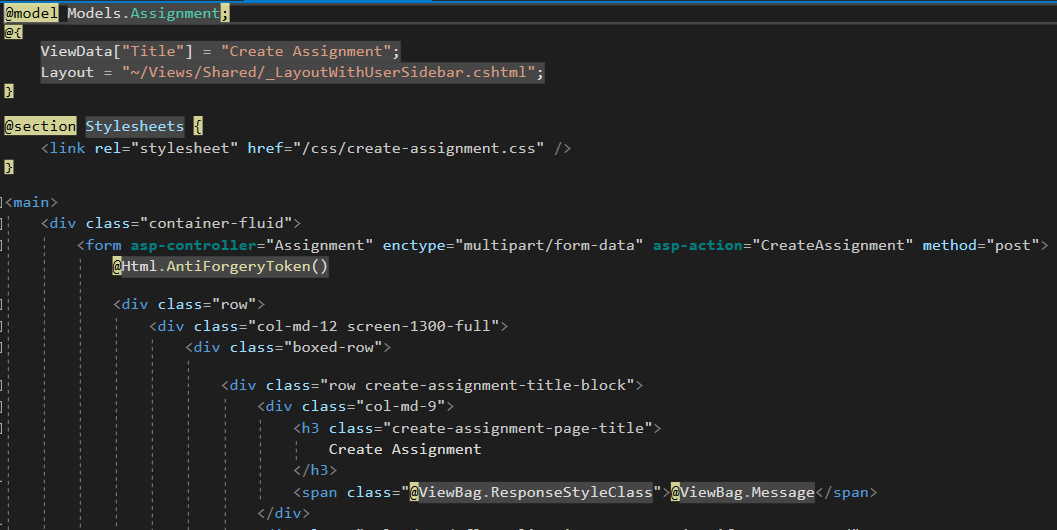


Figure 8 Parametrized dapper query

## **Cross Site Scripting**

Cross Site Scripting also known as XSS is a client-side code injection attack, where the attacker inserts scripts with malicious intent into the website. If successful, the malicious code could be inserted into the websites database and the scripts could then be execute on the original website without the user’s knowledge. Layouts of websites could be modified such as fake login pages, where the user would unknowingly hand in his login credentials to the attacker. The key to take care of this is proper input sanitation. In our case we validate all external input before displaying it on the page, if we expect the input to be for example a number within a certain range, we first validate it on the client side and after that we also validate it on the server side. Razor pages which we use, also offer a layer of protection. The Razor view engine provides default encoding and escaping for inputs and outputs.



**Screenshot of Views file with form and @ HTML encoding**

## **Cross Site Request Forgery**

Cross Site Request Forgery also known as CSRF is a type of attack where the victim user needs to be logged in to a website and a secure session needs to be established at the time of the attack. The unsuspected user will then mostly thanks to social engineering be tricked into clicking on a phishing link with a forged request. This can result into unauthorized money transfers, data theft and changed login credentials. To prevent this there are some useful tools in the ASP.NET MVC which we utilized. Razor automatically generates anti forgery tokens when declaring method=”post” in the HTML form element. We also explicitly added an anti-forgery token with HTML helper @Html.AntiForgeryToken. At last, we added the ValidateAntiForgeryToken in our Views controller for individual POST actions. Including this tag we get validation of the token which has been generated at the creation of the form, verifying the presence of the cookie.

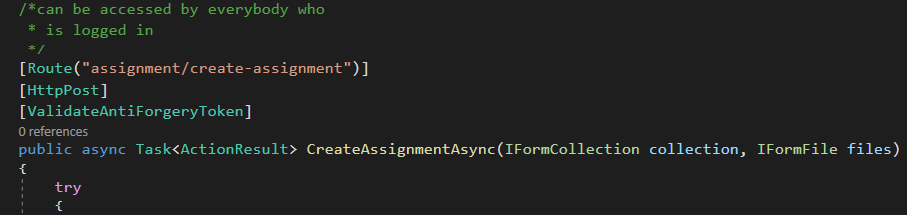


Figure 9 Method with ValidateAntiForgeryToken

## **Brute Force Attack**

Brute Force also referred to as Brute forcing is a type of attack which is based on trial and error. The attacker tries a countless number of forceful attempts to break into accounts or crack encryptions. It is usually carried out through an automated software which executes the task. Depending on the strength of your passwords these attacks can take anywhere from seconds to many years. [2] This is the reason why it is important to use strong passwords and encryption. Our minimal password requirements include at least six characters, upper case character, lower case character, numeric value and an non-alphanumeric value. Based on our password strength requirements it would take up to years for a brute force attacker to find the right combination. If we had more time we would have included a set number of allowed attempts to log into your account until eventually locking the account, another feature would have been 2 factor authentication. These precautions would make it almost impossible for any intruder to use a brute force attack to get into our system.

## **Distributed Denial Of Service Attack**

Distributed Denial of Service also wildly known as DDoS, is a type of attack where the perpetrator sends hundreds of thousands request to a web site, essentially flooding the network traffic and preventing normal users to enter. The website is not able to comprehend such an overwhelming bulk of requests and usually goes down. Dynamic IP Restriction is a way of preventing this kind of attack. It is monitoring IP addresses with suspicious activity based on patterns of behavior. If an IP address of such nature is detected it is blacklisted and denied access for the website. [3]

## **Man In The Middle Attack**

Man In The Middle also known as MITM is a type of attack where the intruder intercepts communication between two parties (eg. user and website), this may lead to manipulation of the connection and theft of sensitive data. It usually happens on public WiFi hotspots where the internet connection is not private and the website visited does not use HTTPS (Hyper Text Transfer Protocol Secure). To prevent this, we use app.UseHttpsRedirection() in our startup file, this way only HTTPS is enforced meaning the secure connection is established and every communication is encrypted using a public key encryption.

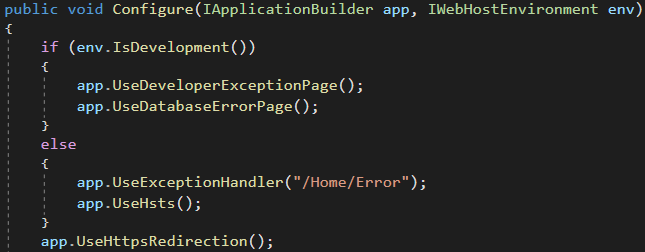


Figure 10 Startup file with app.UseHttpsRedirection

## **Over Posting attack**

Over posting also known as mass assignment, is a type of cyberattack where the attacker is exploiting model binding to a request vulnerability. When a user is binding data to a model by an action like filling up a form, by design not all properties must be visible to him. If not properly secured the attacker is able to set values to properties, he normally would not be able to. [4] Using MVC in our project we were naturally exposed to this vulnerability. Solutions to this would be adding DataAnnotations such as [Editable(false)] to the properties in the model class. This way the data binder would ignore any attempts of editing a specific property.

# **Concurrency**

Concurrency generally is the programs ability to be executed parallelly. This can be encountered for example on the Internet, where two users use the same website at the same time. Concurrent usage of shared resources can lead to different multi-user problems or conflicts, which we tried to took care of.

## **Optimistic vs. pessimistic**

There are two approaches to resolve concurrency conflicts: optimistic and pessimistic. Pessimistic concurrency assumes that the conflicts are going to happen, and often. We thus lock the database, making sure no one will interfere and create a conflict. This of course means that during this lock period, no one will be able to access the database until it is unlocked, which limits the user greatly.

Optimistic on the other hand assumes that conflicts will not happen. The user is let to do what he desires and if he unknowingly creates conflicts, they are resolved afterwards. It does not limit the users in any way, which means it is the best solution for systems with relatively low possibility of conflicts. Also, compared to the pessimistic approach it is often easier to implement. Each approach has its shortcomings.

Pessimistic can run into 'deadlocks' - a state when each resource waits on the other to make some progress, waiting indefinitely. This can lead to the system not responding and being stuck at the same point. Optimistic on the other hand can resolve into 'livelocks' - states like deadlocks, but where each resource is constantly changing its state in relation to others, but without any progress. A good example of a real-life livelock would be when someone comes across a person in a narrow corridor. They both politely try to move aside for the other person but end up going back and forth the same way, without any progress.

## **Transactions**

Transactions are a single unit of various tasks/queries that gets executed. Each task gets executed individually and based on the outcomes, the transaction results in a success (every task was successful) or a failure (at least one task failed). A database transaction should always be ACID - atomic, consistent, isolated & durable. They are often the basis for pessimistic concurrency control, although they can be part of optimistic concurrency as well.

At several places in the code, we tend to use transactions instead of plain Dapper queries. The reason is simple - performance. This means that transactions are used not only to batch SQL queries together which either succeed or are rollbacked, but they are also executed faster, whether the query is an insert or an update. The reason why using transactions is more time-efficient is because the transaction is always implicitly created, for say a write operation. This means that if one specifies when the transaction starts and ends, SQL does not have to figure it out on its own.

(pic for insert)

(pic for update)

It must be noted that naturally, each transaction has an overhead, meaning the less transactions, the better. Having a single transaction instead of ten of them saves a significant amount of time. But there is a downside, if one query fails, the entire transaction is rollbacked, meaning we cannot wrap everything into a single transaction. Another important part plays the forementioned isolation levels. When implementing transactions, we had to make sure that they are not wrongly blocking other queries to the database. When it comes to the performance of individual isolation levels there is not much of a difference. (pic).

## **Our concurrency**

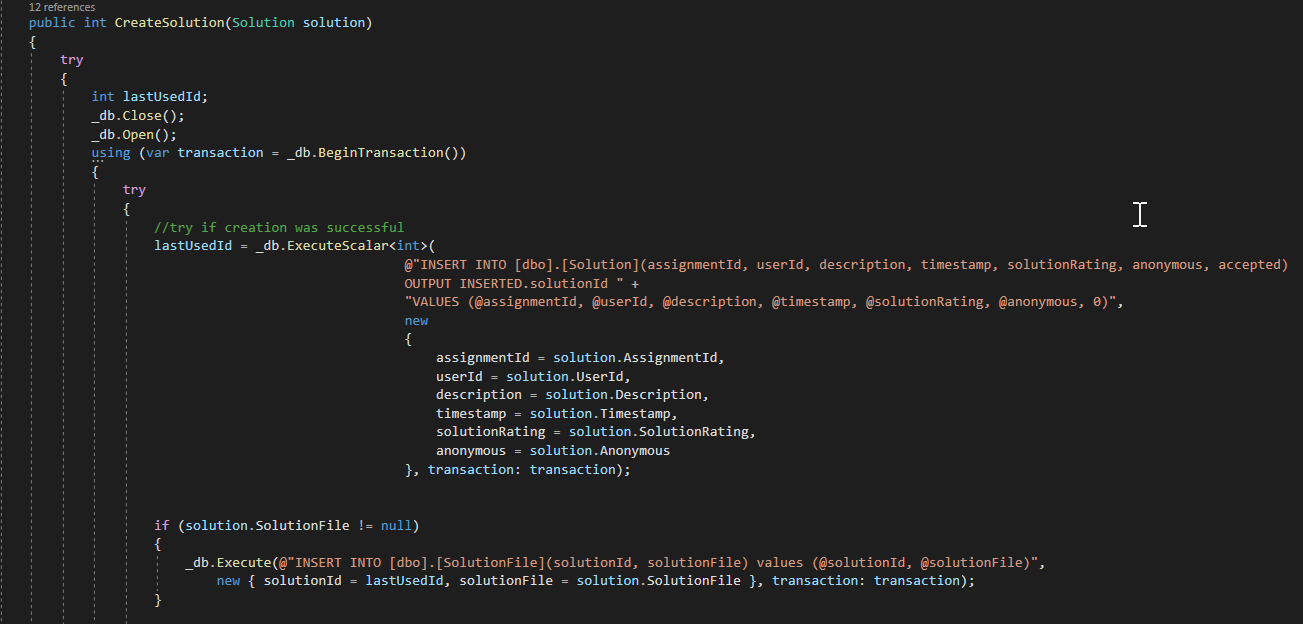
Let us look at how different multi-user issues were solved in our system. For majority of the issues, we chose to use the optimistic concurrency, because it suits our platform the most. There are not that many database updates and we feel like response time is crucial for web solutions so locking the database would at times not be ideal. Of course, if we anticipated that inserting and updating would be crucial for our system, pessimistic solutions would be considered and that is precisely why we use them for specific operations.   
It is important to establish 4 different types of users on our platform:

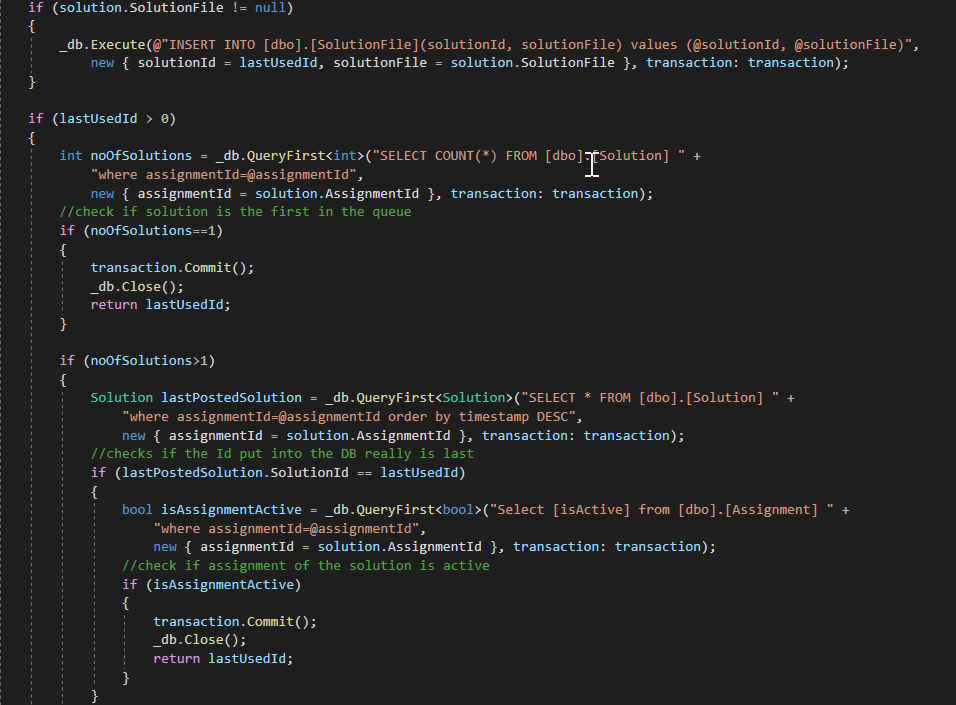
1. a poster - a person who posts an assignment;
2. a solver - a person who provides a solution to the assignment;
3. a basic user - a person who just browses the website;
4. moderator - the only person who would use the dedicated client to moderate the posted assignments/solutions and the forum.

The most common multi-user/concurrency issue we came across are:

**Multiple users trying to post an answer to a solution at the same time**

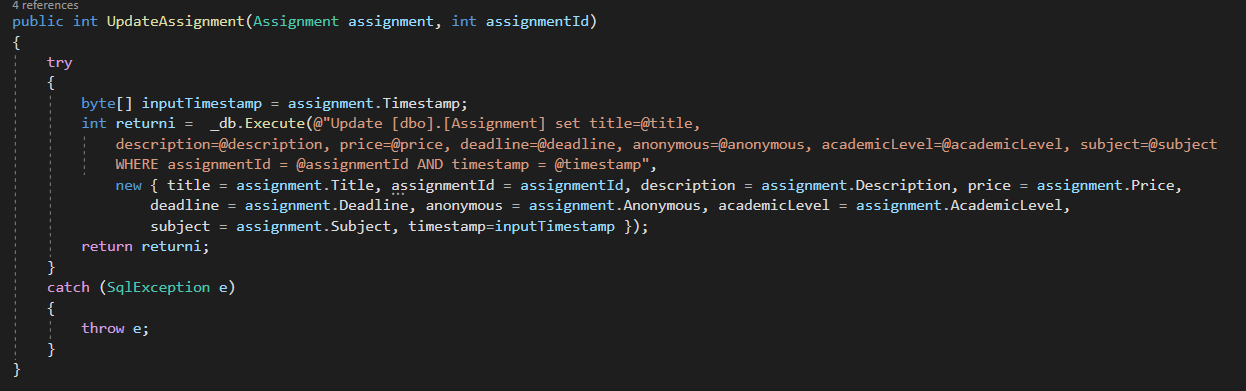
The solution we have come up with is using a pessimistic solution. When user confirms the solution to an assignment, he locks the database for a moment. This would mean that anyone else trying to post a solution at the same time would be prompted to submit his solution again. Like this, there would always be a clear queue of who posted the solution first. Even though this solution locks the database, we do not see a way it would reach a deadlock.





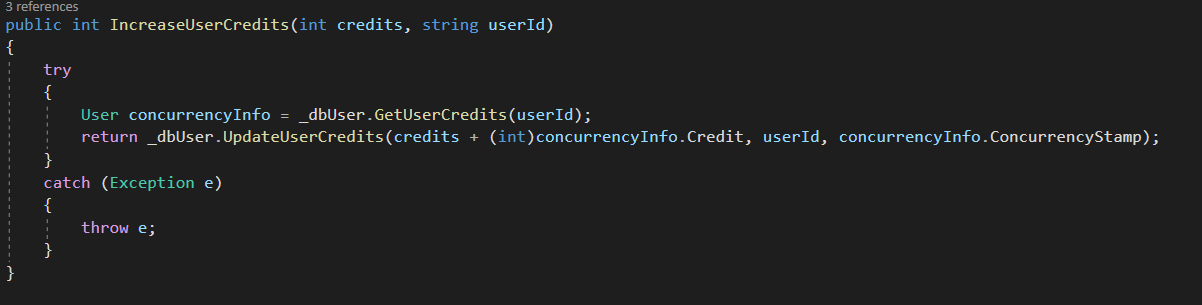
**User is updating the assignment at the same time as the moderator**

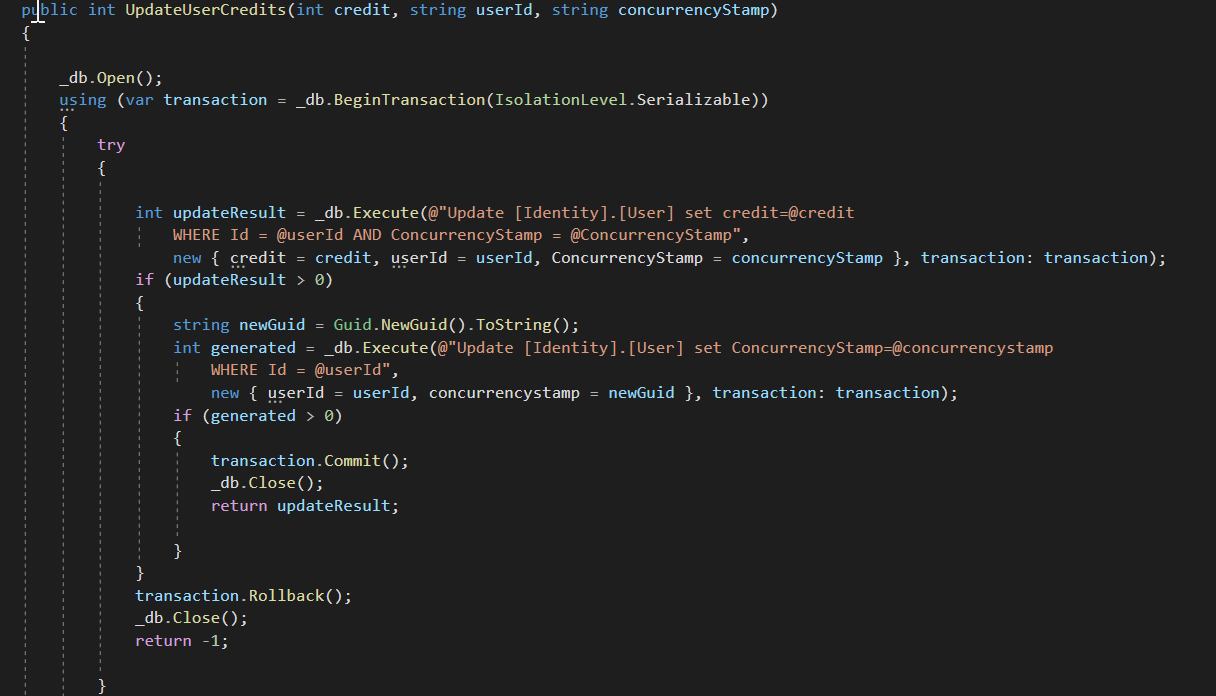
This scenario we decided to solve using a timestamp column of a SQL rowversion type, which is newly generated whenever the assignment in the database is changed. We get the initial timestamp from the Razor page when the user loads it, because that represents the state of information the user is presented with. When he then updates the assignment, we compare this timestamp with the one in database. If they match, the update goes through, but if they do not, the user is prompted to try again, so he is aware of the changes made in the meanwhile.



**User is adding credits to his account and at the same time moderator accepts the credit return request and adds the credits to user’s account**

We decided to solve this problem with a possible lost update again with optimistic concurrency. Every single any credit-related update happens, Users concurrencyStamp is checked. The concurrencyStamp is a GUID (globally unique identifier) that we get before the update to have the latest information. This stamp is compared with the one in the database and if it matches, in a transaction with the update a new concurrencyStamp is generated. If the stamp was changed in the meanwhile, the user is notified to try again.



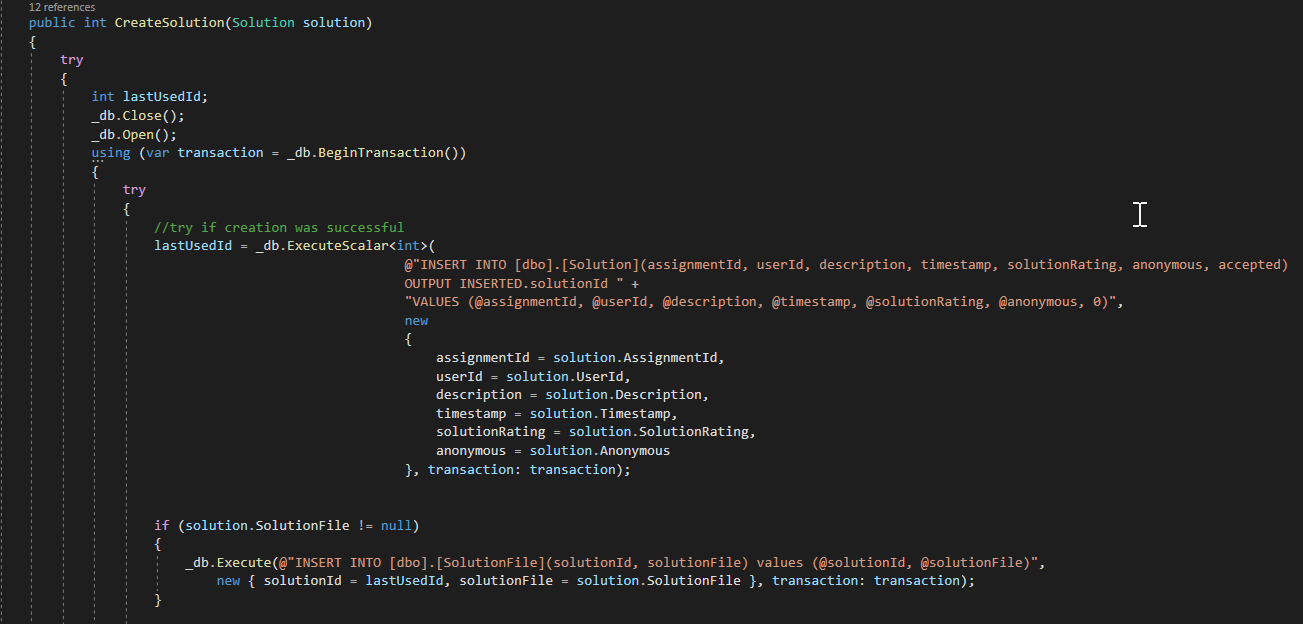


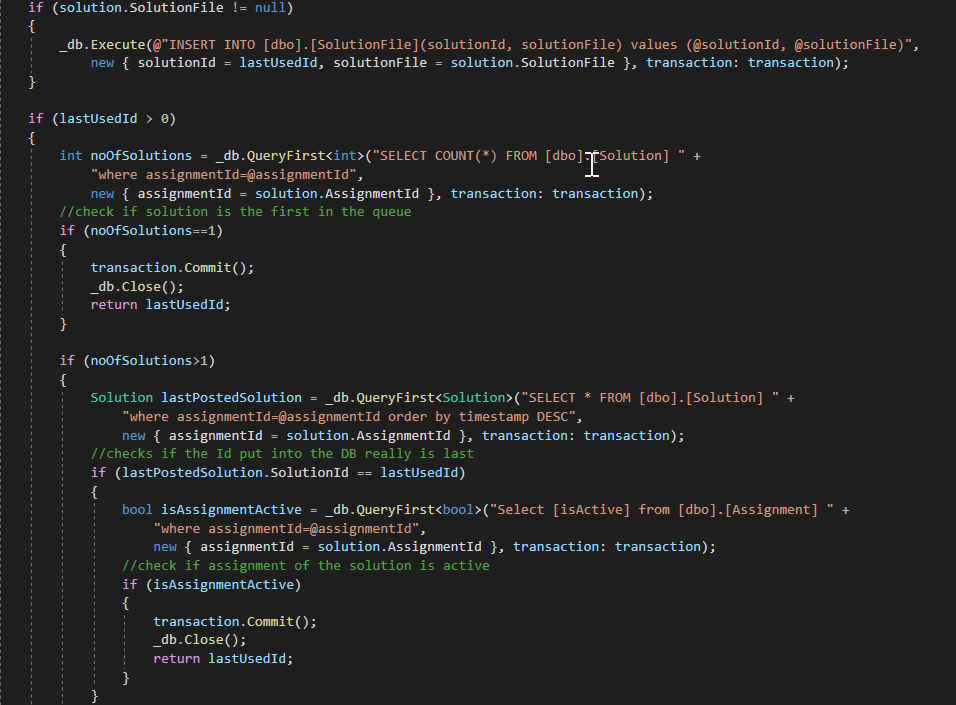
**Poster deleting a post while the solver is answering it, leaving the solver’s solution “hanging up in the air” as there is no post to associate it with**

As previously mentioned, the creation of solutions is done pessimistically. That means that the operation which starts first sooner will finish before the latter is started. Essentially there are two scenarios:

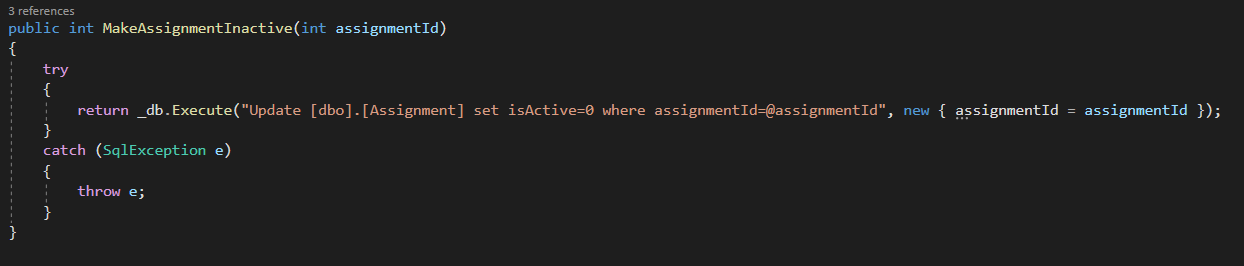
The solver posts a request to the server. The solution has the ID of the assignment attached to it, so the server starts looking for the same ID among the posted assignments. If it finds it, but it is deleted (just disabled, deleted in eyes of the users) or inactive, the solver gets a response from the server, that the assignment has been deleted in the meantime and his solution was therefore not accepted.

If, on the other hand, the server finds an active assignment with a matching assignment ID, it attaches the solution to it, using a transaction for that process), so if there is a delete request from the poster in the meantime, it first lets the solution to be attached and then the post is deleted or disabled. If in fact the post is to be deleted after the transaction that attaches the solution to the post is finished, another transaction is started (for deleting the post) and the post is deleted.





(same pictures as in the first case, maybe redundant???)



**Poster confirming an answer and paying for a solution while the solver is deleting the solution leaving the poster’s payment “hanging up in the air” as there is no solution to associate it with**

This issue has not been solved yet, although we were awfully close. This scenario which could occur when poster goes through all the solutions, decides to choose a one and pay for it with his credits, while the user who posted the solution decides at the same instance to delete his solution, undesired outcomes could happen.

**The solution could be just an extra SQL query in the transaction that would check if indeed the solution the solver receives money for is still active.**

If the solver really would not want for his solution to be accepted, we could solve that by connecting the two parties right after the payment happens, where there would be a chance for them to discuss the solution and solve the possible problems, perhaps by agreeing to a refund.

**Moderator deleting someone’s post for say abusing platform rules while the solver is answering that post, leaving the solver’s solution “hanging up in the air” as there is no post to associate it with**

The solution for this problem is be the same solution we use for the problem where poster deletes a post that is being answered at the same time, as the logic behind it is the same, just the user who is deleting the post is different (poster vs moderator).

**Moderator banning a user while some poster is confirming an answer and paying for a solution which was posted by the banned user, leaving the poster’s payment “hanging up in the air” as there is no solution to associate it with**

The solution for this problem could be the same solution we use for the problem where the solver is deleting him solution, as the logic behind it is the same, just that in this case, the solution is not deleted, but the solver is banned by a moderator, which means none of his assignments or solutions can be used.

Unfortunately, we have not found the time to implement a feature which would enable the moderator to ban users.

**Two users registering with the same credentials at the same exact moment**

We were deciding between optimistic and pessimistic approach. We decided to handle this issue with pessimistic locking by applying Try/Catch construct, it prevents users and applications from editing data that is being or has been changed. Processes know immediately when a locking violation occurs, rather than after the transaction is complete.

# **Testing**

## **Unit testing**

Unit testing is the initial testing phase. It consists of testing the smallest components or modules that make up the whole system. If done correctly and early enough, the development team can single out where the issue occurs and easily, without troubles integrate more individual code modules. Unit tests are usually automatic and should be run before one start working on a feature for example. If he does not, the time and resources used to look for the bug that potentially could be somewhere in the system could outweigh unit testing significantly.

Typically, a unit test is made of three phases - Arrange, Act, Assert (known as AAA). In the Arrange phase, we set-up the test (e.g., we prepare and initialize all the variables). Later, in the Act phase we do the actual test. Its result is then checked in the Assert phase where the test either returns true (passes) or false (fails).

To unit test our code, we used the commonly used NUnit framework. Overall, unit tests were a significant part of our source code. They were covering each of the model class and they made sure the methods used in the Data Access Layer (DAL) were working flawlessly. During the development process, it was often hard keeping the tests up to date due to various changes in the database or the model layer. It must be admitted we did not run these tests as often as we should have, especially towards the end of the later sprints, where the deadline was approaching fast.

Even though we did not find the time to test the entire DAL during the development, we made sure to add the tests afterwards, so that someone else can pick up the code and easily work with it.

# **Code standards**

Programming languages are for humans, computers suffice with just ones and zeros, so let us make this as pleasant for us as possible. For a better readability of the code, we felt that it is important to follow the right Code Standards from the beginning of the development process.

**What we did**

1. Naming conventions

* Overall, we used 2 naming conventions, PascalCasing and camelCasing.
  + We used PascalCasing for the naming of methods, classes, public properties, and constants. Since we all have a background in JAVA where camelCasing naming conventions are heavily popularized for methods and properties, it took us some time to get used to it.
  + On the other hand, we were familiar with using camelCasing for local variables and method arguments.
* We would always try to achieve clear and brief naming of variables and methods based on their functionality and purpose. (eg.FindSolutionByAssignmentId, FirstName)
* Interfaces in our repository layer would be always named with a capital I at the beginning of the name (eg. IDBAssignment).
* Source files were named accordingly to their Class name.
* Private properties would always start with underscore (e.g., \_connection)

1. Declaration

* Declaring all class members at the top of the class
* Declaring local variables close to their usage

1. Indentation

* The right number of white spaces between complex if statements and method calls for easier readability.
* White spaces between initializations
* Vertical alignment of curly braces {}
* Using tab key for line spacing

1. Consistent return values for methods

* When returning error values from integer methods we would commonly use the number -1.
* Error status codes for the web client (400,404,405)

**What we avoided**

* Reusing names of variables which have different purposes
* We avoided type identification in the naming of variables and methods. (e.g., strValue)

# **Implementation**

* Used data structures and why chose it
* Code snippets
* Interesting code
* WPF for GUI
* Data formats

## **Pagination**

The pagination is an important aspect when talking about user experience, it helps the user to better visualize lists of objects. The more important effect of pagination is loading time. When trying to load a lot of data from the database, it might take a while for it to appear on the frontend. To avoid that, we try to cut the data into chunks, and only load a specific amount. In our project, we are using pagination when displaying the assignment cards. We decided that 12 assignments per page would be the ideal number because the page can have either 1,2 or 3 assignment cards in a row, depending on the size of the screen. When trying to display any page containing assignment cards, the request also sends to the API the page number that we are trying to see, the default page being 1. Then, the API reaches to the database and counts all the rows that contain the information we are looking for. If the number of rows is bigger than the previous page times 12, then it returns a list of 12 assignments starting from (PageNumber - 1) \* 12 + 1. Included in the HTTP response header is also the page metadata, it tells the web client the total number of pages, how many assignments there are in the database that match the request, but also if the previous page and the next page exist.

# **CONCLUSION**

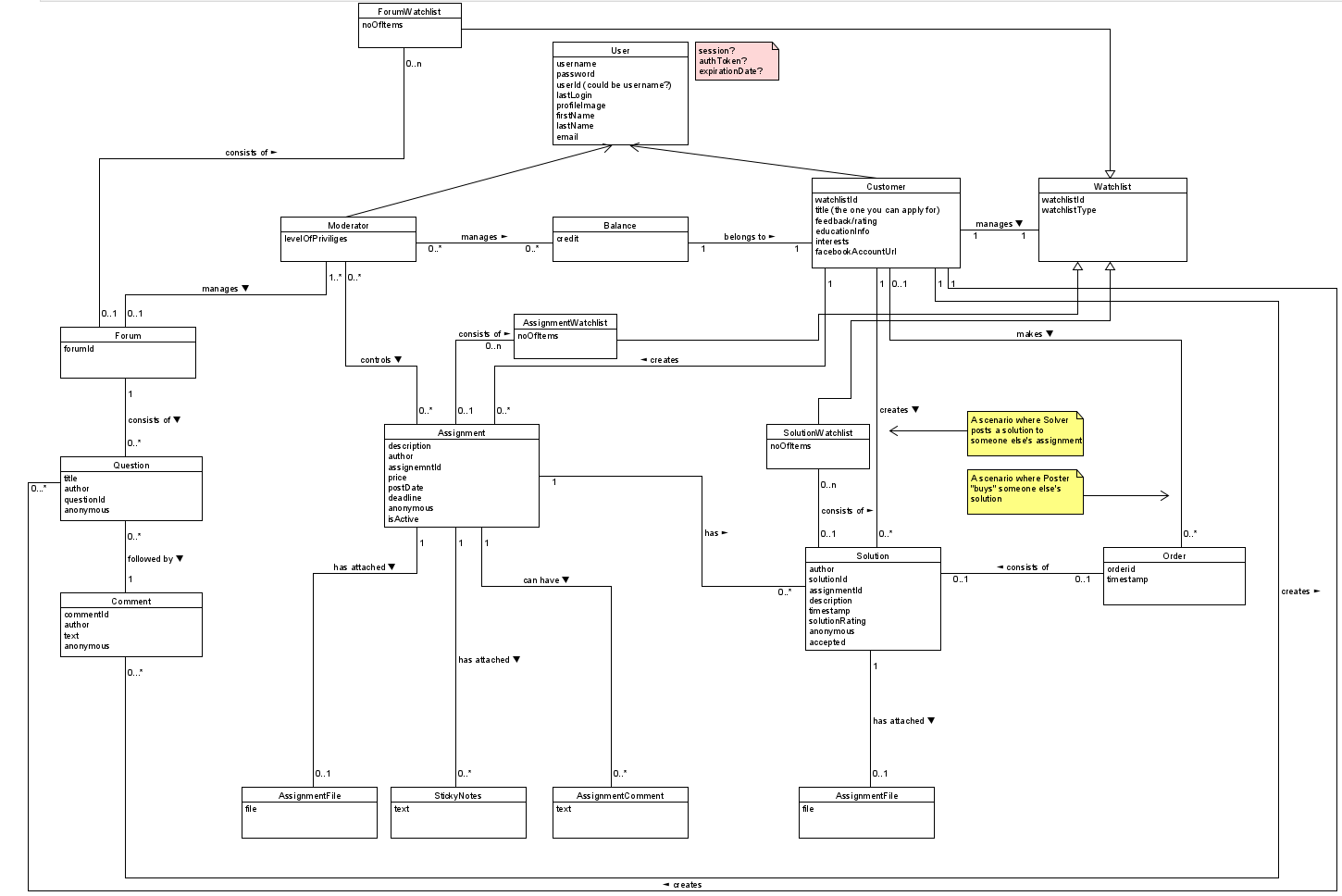
# **REFERENCES**

|  |  |
| --- | --- |
| [1] | "Dapper Tutorial," [Online]. Available: https://dapper-tutorial.net/dapper. |
| [2] | "Brute Force Attack: Definition and Examples," Kaspersky, [Online]. Available: https://www.kaspersky.com/resource-center/definitions/brute-force-attack. |
| [3] | "Dynamic IP Restrictions," Microsoft, [Online]. Available: https://www.iis.net/downloads/microsoft/dynamic-ip-restrictions. |
| [4] | "Preventing mass assignment or over posting in ASP.NET Core," NET Escapades, 21 3 2017. [Online]. Available: https://andrewlock.net/preventing-mass-assignment-or-over-posting-in-asp-net-core/. |
| [5] | "Making Dapper Faster with Transactions," 26 4 2012. [Online]. Available: https://blog.staticvoid.co.nz/2012/making\_dapper\_faster\_with\_transactions/. |
| [6] | "GeeksForGeeks," 14 8 2020. [Online]. Available: https://www.geeksforgeeks.org/sql-transactions/. |
| [7] | "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Deadlock. |
| [8] | "Optimistic Concurrency vs Pessimistic Concurrency – short comparison," 23 2 2017. [Online]. Available: https://agirlamonggeeks.com/2017/02/23/optimistic-concurrency-vs-pessimistic-concurrency-short-comparison/. |

# **APPENDIX**

# 

# Appendix A – Domain Model



# Appendix B – Relational Model

Chart, bar chart

Description automatically generated