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

Project participants:

Maros Cuninka

Sebastian Holesz

Samuel Horacek

Martin Hotka

Ioan-Sebastian Voinea

Supervisor:

Nadeem Iftikhar

Abstract:

This is a report documenting the technical process that our group conducted during the 6 weeks prior to the hand in on the 21st of December. On the following pages, you will be able to read about our implementation of conceptual classes in the domain model, using database to store all information, through the Restful API up to desktop and web clients. We will also explain how we handled concurrency, authentication & authorization, and what security issues we have considered as important in our scope. Last part is focused on the implementation.

Submission date: 21. 12. 2020

Contents

INTRODUCTION

Our platform Solvr.online aims to connect people who need help solving their assignment or homework with potential solvers, whilst both sides benefit. The poster gets the help needed and the solver gets some financial compensation. This report should serve as technical documentation for this system. We will take you through the individual blocks on which this idea of a system is built upon while explaining our options and conclusions. Firstly, on the theoretical level with the domain model, system architecture, choice of API, communication protocols, and others. Following is the actual implementation - how we handle concurrency conflict, various security risks, testing and at the end, we show interesting features or code we are proud of.

1. Domain model

Even though agile methodologies suggest, to avoid inevitably outdated artifacts, that no or only minimal documentation should be done, the development team decided to create an initial domain model of the system.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 6 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 (*Appendix A*). 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 added the multiplicity (*Figure 1*).

The most important classes are assignment and solution. An assignment can have multiple solutions, while 1 solution is assigned for 1 specific assignment. Initially, we applied inheritance for users since we were dealing with customers and moderators. They share the same properties in the superclass but have their own as well. This was changed by implementing identity through the MVC but we decided to keep it in the Domain Model for better visibility a further implementation. Each of them has different privileges. The customer can either post an assignment or a solution and can add credits to his account. Moderators can see all assignments, even inactive, solutions, add and remove credits from the user's balance. In the next sprints we would want to implement a forum for users, where they could ask different questions, and a watchlist for assignments, so the user gets a notification if the assignment he follows changes.

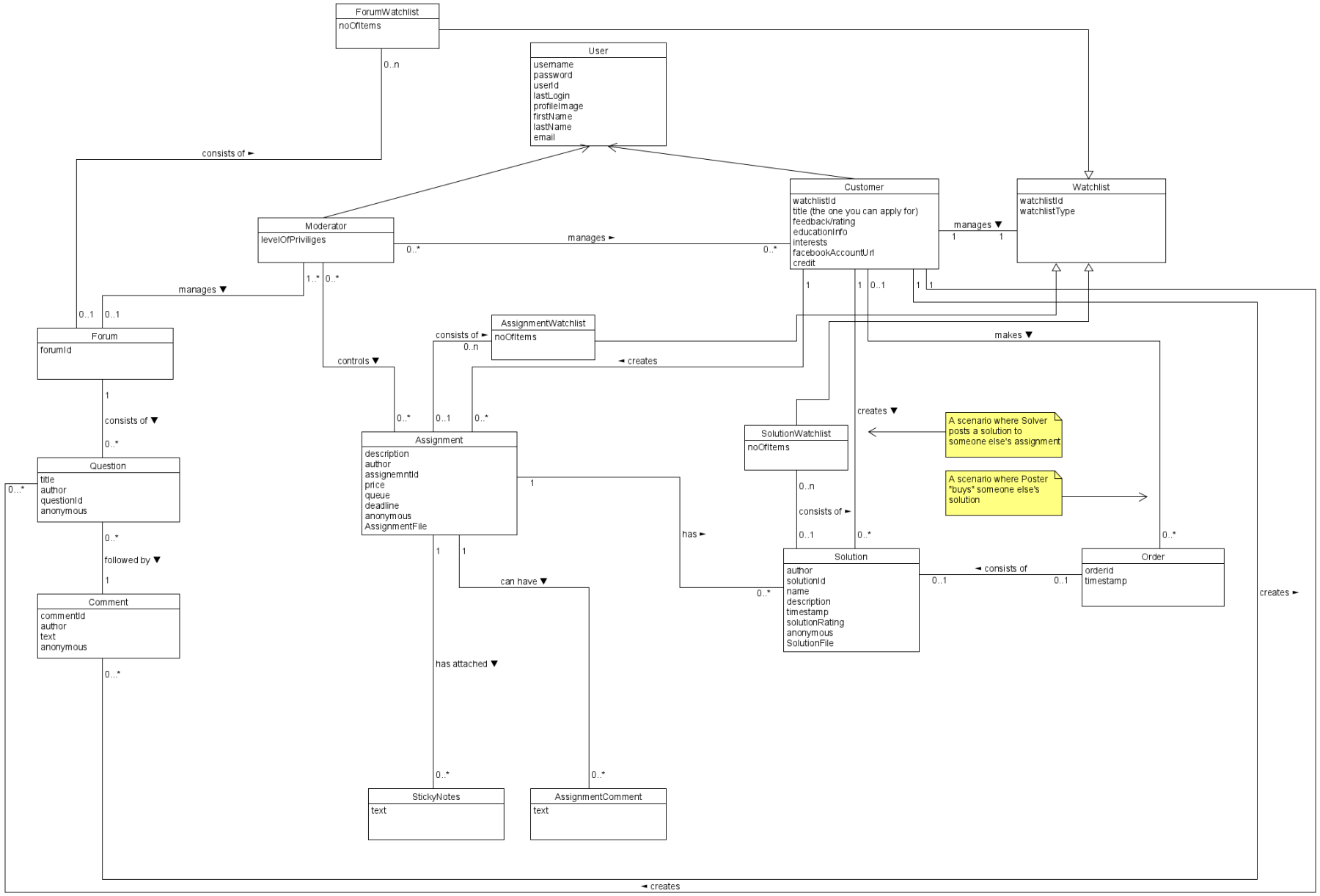

Figure 1 Domain Model

2. Database design

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

2.2. Database first & Coding first

When creating a system that is also connected to the database, developers can choose between 2 approaches – code first and database first.

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

In the Database first approach, developers firstly create the database with tables, which are used to create an entity data model. The main advantage of this approach is mapping and creating keys and relationships between relations. Another advantage can be a graphical user interface while creating tables, e. g. in MSSQL Management Studio. [2]

In our project, we used both approaches. Since we were using dapper, which only maps an existing database, we had to use the database first approach, for getting data about Assignments and Solutions from the database. By implementing authentication & authorization through the individual user accounts provided by MVC we used the entity framework.

We created tables in the code and then by using migrations we updated the database. The same logic was applied for inserting new columns into the identity table (firstName, lastName, profilePicture for User).

3. Architecture design

The architecture refers to the fundamental structure of the system, it serves as a blueprint on which the system is developed. A well-laid architecture helps to set the development team on the same track. It is also a good way of communicating with a customer, as it exposes the structure of the system, but it hides the implementation details.

There are multiple types of architectures that developers use when building software systems.

N-Tier Architecture

For this project, we used multi-tier architecture, also known as n-tier architecture. The core feature of this type of structure is that it does not only logically separates the processing, data management, and presentation functions, but also physically. Meaning that these different layers are hosted on several machines, thus, making them tiers. It ensures that the services are provided without resources being shared, this way becoming easier to manage, working on one section, not affecting the other. This added flexibility can also improve the overall time-to-market decreasing development cycle times.

3-tier

A 3-tier is a type of multi-tier architecture that contains 3 different sections of logical computing. It modularizes the user interface, business logic, and data storage layers. In other words, the 3-tier architecture is split into Presentation Tier, Application Tier, and Data Tier (*Figure 2*).

The presentation tier is the front-end layer of the system and consists of the user interface. It can be represented by one or more clients, such as web client, dedicated client, or mobile client. The application tier consists of the business logic which is responsible for the product's core capabilities. The data tier incorporates the database or the data storage system. The presentation and data tiers communicate only with the application tier and not with each other.

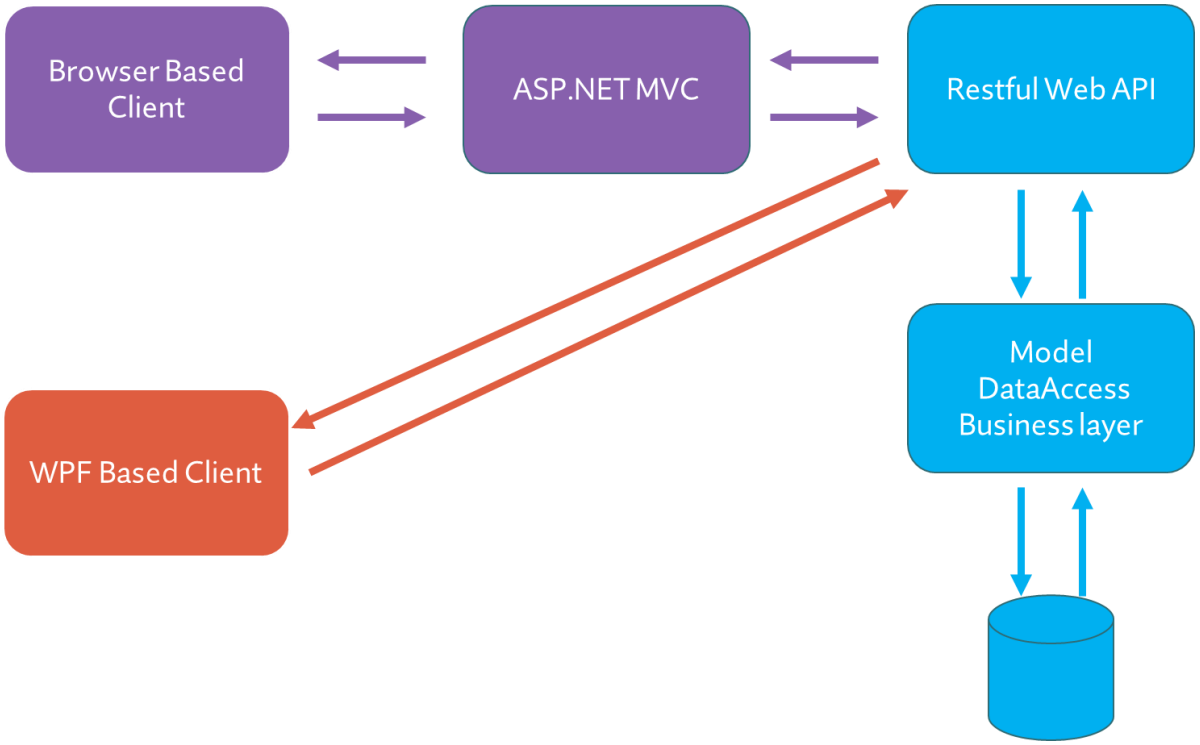

Figure 2 Our architecture design

The 3-tier architecture has a couple of benefits that made us consider it as our choice. Firstly, it provides a great degree of freedom to the team, enabling us to work independently, updating or replacing, only specific parts of the system. The application can be also easily scaled up or out, and new hardware or technologies can be added later.

As with any architecture, it also has disadvantages, the biggest one from the development point of view is debugging. Although the structure may be simple to visualize and understand, it makes the debugging process rather difficult and complex. Another disadvantage represents the fact that implementing even small parts of an application will consume lots of time.

3.1. **Communication protocols**

Protocols are sets of rules that must be abided when two devices communicate with each other. They are required, if it would not be for the protocols 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) & 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 sender's message, divides it into smaller parts (packets), and sends them across the network. The receiver reassembles the packets together and gets the message. TCP makes sure that the receiver got all the packets, in the same sequence, without duplications or damages. This makes it slower in comparison to UDP (User Datagram Protocol).

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. It is not reliable, because it does not ensure it reaches its destination. 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 have a specific port on which it listens to responses from the network.

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

3.2. REST vs SOAP

Choosing the right API architecture is very important. There are two frontrunners in the API industry, which are as similar as different.

REST, short for Representational State Transfer, is an architectural style that delivers APIs through HTML. It heavily utilizes the URL to make data available using the web. Rest can use many data formats such as JSON, XML, YAML, and even plain text. Compared to SOAP, Rest is faster, more flexible, and very data-driven. It is widely used in mobile development and single page web applications.

SOAP short for Simple Object Access Protocol is a protocol that relies on schemas to exchange messages between systems. These schemas are explicitly defined, using only XML

as the data format. SOAP is slower than REST, more restricted, and tightly coupled with the server, it is widely used for enterprise solutions (e.g. Banking), where security is very important. [3]

From our perspective, it was an easy choice. REST is lightweight and flexible which perfectly fits our purposes. Using REST did not bind us to one data type. Our data format of choice was JSON, which is lighter and smaller than XML, which would be required to use when choosing SOAP.

3.3. Clients

3.3.1. Web Client

Single Page Application and Multi-Page Application

To develop a web client, multiple choices regarding patterns and architectures must 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.

Implementation

As you can see in the picture below (*Figure 3*), when a user wants to create a new assignment, the button "Post Assignment" redirects him to the "/assignment/create-assignment" route.

```
<a href="/assignment/create-assignment" class="btn btn-post-assignment">
    <img class="btn-post-assignment-icon" src="/assets/icons/add.svg">
    Post Assignment
</a>
```

Figure 3 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 *figure 4*, you can see how the "Create Assignment" method makes HTTP requests to the Web API 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.

```
[Route("assignment/create-assignment")]
[HttpGet]
0 references | sebaholesz, 4 days ago | 4 authors, 20 changes
public ActionResult CreateAssignment()
{
    try
    {
        using (HttpClient client = new HttpClient())
        {
            User user = _userManager.FindByIdAsync(User.FindFirstValue(ClaimTypes.NameIdentifier)).Result;
            client.DefaultRequestHeaders.Authorization = AuthenticationController.GetAuthorizationHeaderAsync(_userManager, _signInManager, user).Result;

            string urlGetAllAcademicLevels = "https://localhost:44316/apiV1/academiclevel";
            string urlGetAllSubjects = "https://localhost:44316/apiV1/subject";
            string userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
            string urlGetUserCredit = "https://localhost:44316/apiV1/user/get-credit";


            HttpResponseMessage academicLevelsRM = (client.GetAsync(urlGetAllAcademicLevels).Result);
            HttpResponseMessage subjectsRM = (client.GetAsync(urlGetAllSubjects).Result);
            HttpResponseMessage urlGetUserCreditRM = (client.GetAsync(urlGetUserCredit).Result);


            if (academicLevelsRM.IsSuccessStatusCode && subjectsRM.IsSuccessStatusCode && urlGetUserCreditRM.IsSuccessStatusCode)
            {
                ViewBag.AcademicLevels = JsonConvert.DeserializeObject<List<string>>(academicLevelsRM.Content.ReadAsStringAsync().Result);
                ViewBag.Subjects = JsonConvert.DeserializeObject<List<string>>(subjectsRM.Content.ReadAsStringAsync().Result);
                ViewBag.Credits = JsonConvert.DeserializeObject<int>(urlGetUserCreditRM.Content.ReadAsStringAsync().Result);
                return View("CreateAssignment");
            }
            else
            {
                throw new Exception("Internal server error");
            }
        }
    }
    catch (Exception e)
    {
        if(e.InnerException is UnauthorizedAccessException)
        {
            return Unauthorized();
        }
        TempData["ErrorMessage"] = e.Message;
        return Redirect("/error");
    }
}
```

Figure 4 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 5*).

```
<div class="form-group">
    <label for="AcademicLevel">
        Academic Level
    </label>
    <br />
    <select asp-for="AcademicLevel" class="form-control">
        <option value="" selected>Choose academical level</option>
        @foreach (var option in ViewBag.AcademicLevels)
        {
            <option value="@option">@option</option>
        }
    </select>
    <span asp-validation-for="AcademicLevel" class="text-danger"></span>
</div>

<div class="form-group">
    <label for="Subject">
        Subject
    </label>
    <br />
    <select asp-for="Subject" class="form-control">
        <option value="" selected>Choose subject</option>
        @foreach (var option in ViewBag.Subjects)
        {
            <option value="@option">@option</option>
        }
    </select>
    <span asp-validation-for="Subject" class="text-danger"></span>
</div>
```

Figure 5 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 with their length or range. 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, we use the "_LayoutWithUserSidebar" which includes the navigation bar at the top of the page, and the user sidebar on the left side for almost all our pages.

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 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.

3.3.2. 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.

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 an older concept for developing desktop applications, but on the other hand, are easier to use. WPF provides better resizeability as they are not pixel-based unlike Windows forms. The last difference to mention is performance, in WPF things are achieved at a faster rate compared to Windows forms. [4]

For our project, we were working with one main window containing a frame with various pages:

- Login page,
- Home page (for all assignments, solutions, and users),
- Update page for an 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. We then chose which component we want to use (button, textbox, textblock…) and placed it into a specific row and column. With this, we achieved good resizeability as well.

The main focus with the WPF was not the design itself, but the connection with API. For this, we created an 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. This way, we could get all assignments or users from the database, update them, or do other operations. One struggle we had, was to implement the authorization, since desktop application was supposed to be used only by moderators. We dealt with it by creating a bearer token when a user presses the login button, sending his username and password to API where it is checked in the database (*Figure 6*).

```
public static bool Login(string username, string password)
{
    using HttpClient client = new HttpClient();
    string urlCreateJWT = "https://localhost:44316/apiV1/login-admin";
    HttpResponseMessage createJWTRM = client.PostAsync(urlCreateJWT, new StringContent(JsonConvert.SerializeObject(new { username, password }), Encoding.UTF8, "application/json")).Result;

    if (createJWTRM.IsSuccessStatusCode)
    {
        dynamic tokenObject = createJWTRM.Content.ReadAsAsync<object>().Result;
        string token = tokenObject.token;

        if (!token.Equals("") && token.Length > 0)
        {
            Logintoken = token;
            return true;
        }
    }
    return false;
}
```

Figure 6 Method for creating bearer token

To achieve higher security, all API calls moderators could use, were checking, if the user who is trying to call them is really a moderator. Therefore, for the HttpClient we set the Authorization as request header with his bearer token (*Figure 7*).

```
public class ApiAssignment
{
    3 references
    public static IEnumerable<Assignment> GetAllAssignments()
    {
        HttpClient client = new HttpClient();
        client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
        client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", ApiAuthentication.Logintoken);
        HttpResponseMessage response = client.GetAsync("https://localhost:44316/apiV1/assignment").Result;
        if (response.IsSuccessStatusCode)
        {
            client.Dispose();
            return response.Content.ReadAsAsync<IEnumerable<Assignment>>().Result;
        }
        else
        {
            client.Dispose();
            return null;
        }
    }
```

Figure 7 Communication with API

To display information about assignments and users we used the DataGrid component. We used one DataGrid for both to achieve better readability and maintainability, the

data is overridden. The same goes for some buttons and text blocks. Thanks to using WPF, we could also apply different styles for components – rounded corners for buttons, background color or hover color (*Figure 8*).

```
<Button Name="ButtonSignOut" Content="Log Out" Grid.Row="8" Grid.Column="1" HorizontalContentAlignment="Left" Click="ButtonSignOut_Click">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Background" Value="Red"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="{x:Type Button}">
                        <Border Background="{TemplateBinding Background}" BorderBrush="Transparent" CornerRadius="20">
                            <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Foreground" Value="White"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
</Button>
```

Figure 8 Applying styles for buttons

4. Authentication & Authorization

Authentication is the process of identifying users and validating who they claim to be. One of the most common and obvious factors to authenticate identity is a password. If the username matches the password credential, it means the identity is valid, and the system grants access to the user [5].

On the other hand, authorization happens after a user's identity has been successfully authenticated. It is about offering full or partial access rights to resources like the database, funds, and other critical information to get the job done [5].

Now that we know what the formal difference is, let us describe our authentication & authorization scheme. As you can see in *Figure 9* (Description in *Appendix D*), we addressed the authentication and authorization problem on multiple levels. We use both JWT (JSON Web Token) and Cookies for accessing different resources.

The easiest way to explain how our users can access our website is a specific scenario of updating an assignment, which is complex enough to show almost every security layer we implemented.

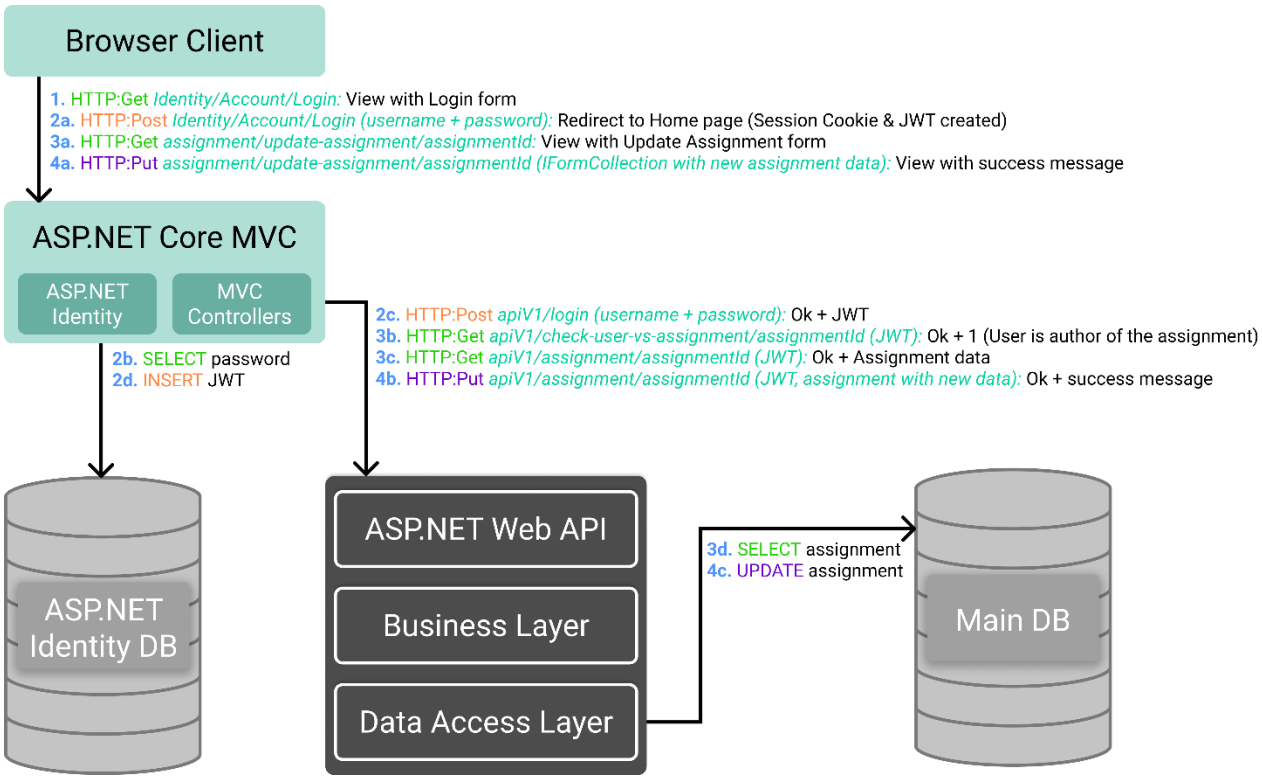

Figure 9 Authentication scheme

In this scenario, you could see that all layers are secured from unauthorized attempts. If a user tries to access resources, he is not allowed to, he will be served an Unauthorized status code. We secured the API endpoints. Every endpoint has a designated user, such as Poster (the user who posted the assignment), Solver (the user who solved the assignment), Not-Solver (the user who has not solved the assignment yet). Those users are different for every assignment, therefore this must be handled in the API. In *Figure 10* you can see an example of an API endpoint that can only be accessed by the Poster of the assignment.

```
/*ONLY AUTHOR*/
[Route("solution/by-assignment/{assignmentId}")]
[HttpGet]
0 references | sebaholesz, 5 days ago | 2 authors, 5 changes
public IActionResult GetSolutionsByAssignmentId(int assignmentId)
{
    try
    {
        string userId = APIAuthenticationController.GetUserIdFromRequestHeader(Request.Headers);
        bool isAuthor = AssignmentBusiness.GetAssignmentBusiness().CheckUserVsAssignment(assignmentId, userId) == 1/*1 means the user is the Poster*/;
        if (isAuthor)
        {
            List<Solution> solutions = SolutionBusiness.GetSolutionBusiness().GetSolutionsByAssignmentId(assignmentId);

            if (solutions.Count > 0)
            {
                return Ok(solutions);
            }
            else
            {
                return NotFound("Solutions with that AssignmentID not found!");
            }
        }
        return Unauthorized();
    }
    catch (Exception)
    {
        return StatusCode(500);
    }
}
```

Figure 30 Authorization in API

Cookie vs Token-Based Authentication

In our application, we use both Cookies and JWTs to authenticate users. We use the Cookie to access the web application. In this case, the user does not know much of what is going in the background. The authentication is done by the server and everything is stored server-side (*Figure 11*). This is not very great for scaling, because if many users try to access resources at the same time, it requires a lot of processing from the server. On the other hand, cookies are usually smaller in size [6], therefore it is faster to send them as a part of the request. It also has restrictions when it comes to domains, as cookies cannot work across multiple domains. Thus, we decided to battle this issue with JWT, which we use to authenticate in our web API.

Our main reason for using Cookies in our web app is that it is simpler to set up the "automatic login" or "login between sessions" as the cookie is stateful. There are also ways

how to achieve this with JWT, such as with the refresh tokens, but we decided to use cookies for our web app and JWT for the API. Looking forward, we would maybe find a mechanism of storing JWT in the browser's local storage and refreshing it using a refresh token.

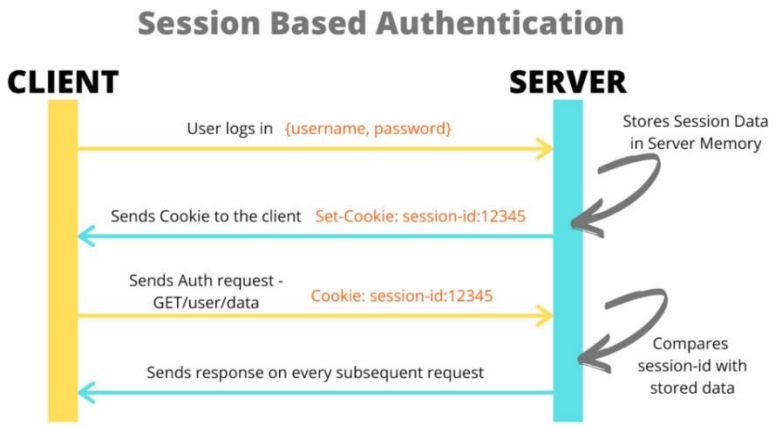

Figure 11 Session-Based Authentication

JWT

JWT is a compact and self-contained way for securely transmitting information between parties as a JSON object. This way we can send encrypted information across the internet to authorize users to access secured resources [7]. JWT consists of 3 parts:

- Header (contains the issuer, the encryption algorithm, etc.)
- Payload (contains the data being transmitted)
- Signature (is a special mean of security protecting the JWT)

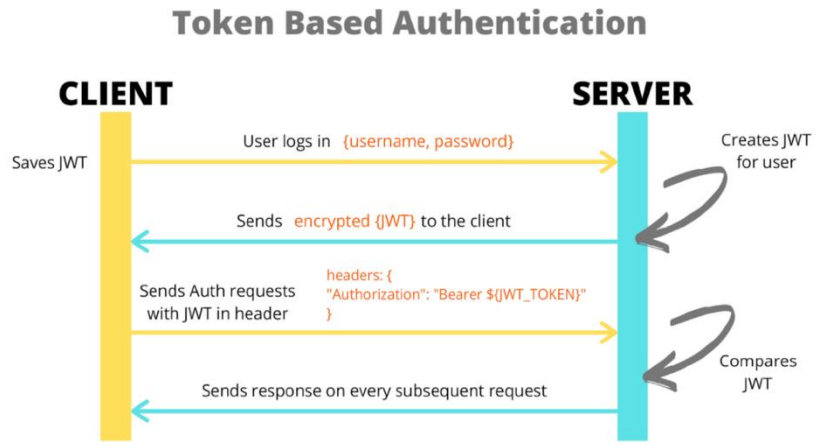

Figure 42 - Token-Based Authentication [8]

4.1. Our implementation of JWT

We decided to use JWT as a bearer token to authorize users to access our API. It is also the only mean of authentication when accessing the API from the WPF desktop application. We then store it in a static read-only field, where the user (admin) can access it at any time and send authorized requests to the API.

Even though the API is just an internal one, therefore not being open to the public, users can always access it using services such as Postman if they know the endpoint's URLs. JWT is a great technology for API authentication and server-to-server authorization [9]. In *Figure 13* you can see the "apiV1/login" API endpoint, which is the default endpoint to authenticate a user. Inside this method, we first take the username and get the needed user data, such as the Id of a user, from the DB. We then authenticate the user by comparing the password hash from the DB with a hash from the incoming password. If successful, a new JWT is generated and sent as a response.

```
[Route("login")]
[AllowAnonymous]
[HttpPost]
0 references | sebaholesz, 6 days ago | 1 author, 1 change
public IActionResult Login([FromBody] User loginUser)
{
    try
    {
        IActionResult response = Unauthorized();

        User loginUserFromDB = UserBusiness.GetUserBusiness().GetUserByUserName(loginUser.UserName);
        loginUserFromDB.Password = loginUser.Password;

        if (UserBusiness.GetUserBusiness().AuthenticateUser(loginUserFromDB))
        {
            var tokenString = GenerateJSONWebToken(loginUserFromDB);
            response = Ok(new { token = tokenString });
        }
        return response;
    }
    catch (Exception)
    {
        return StatusCode(500);
    }
}
```

Figure 53 Login endpoint in API

If a user accesses the API through the MVC web application, his JWT is saved into the Identity DB so he does not need to generate a new JWT with every request. We also made

sure that if a new JWT is generated for a user that already has one, the previous JWT is overwritten, thus making sure the user does not have more than 1 JWT in the DB at any time.

On the other hand, if a user accesses the API through services such as Postman, the JWT is not saved. This means that the user must save the JWT himself. This way we made sure that the "login" and "login-admin" routes, which do not require you to be authenticated but still result in DB reads and inserts, are less vulnerable to overposting.

In some situations, when the MVC web application does not find the JWT in the Identity DB, it tries to create a new JWT before throwing an error. To do that, it cannot authenticate the user with the password, because it does not have the unhashed password outside of the login flow. To handle this scenario, we implemented a SecurityStamp, which is a randomly generated byte array, that changes every time a user changes any of his authentication credentials. With this SecurityStamp and the user's ID, we authenticate the user, after which we generate a new SecurityStamp, so one stamp cannot be used multiple times.

How we generate the JWT

As explained before, a JWT consists of 3 parts (header, payload, signature). Here we will focus on the 2 latest parts.

Our payload consists of 3 claims. We have the user's username, his ID, and a JTI. The JTI is the ID of the JWT. It helps with preventing the JWT from being replayed or reused. The user's ID is what we use for authorization. We use this claim when authorizing users to access certain resources, such as assignments he created, or solutions he posted. We have thought of adding some extra claims to our payload, such as information about the user's login, his user role, or information about his credits, but we simply did not have a use case for it now.

Possibly the most interesting thing about the JWT is the signature. To create the signature part one has to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that [10]. Right now, we are using symmetric encryption, which uses a single public key that must be shared between the sender and the receiver. The key is usually short, typically 128 or 256 bits, that is also why this method is fast in execution. This method of encryption is older and less secure than asymmetric encryption, which uses a public-private key pair and therefore solves the issue of key sharing [11]. The asymmetric key is also considerably longer, often around 2048 bits. It makes this method slower in execution,

but more secure and is better for scaling, because the public key can be shared freely, as the private key is what allows the receiver to decrypt the incoming message. We considered using both methods, but we decided to use the symmetric one, as it is faster both in execution and implementation. We would consider switching to asymmetric encryption if we had to scale our services.

```
private string GenerateJSONWebToken(User userInfo)
{
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

    var claims = new[] {
        new Claim(JwtRegisteredClaimNames.Sub, userInfo.UserName),
        new Claim(JwtRegisteredClaimNames.NameId, userInfo.Id),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

    var token = new JwtSecurityToken(_config["Jwt:Issuer"],
        _config["Jwt:Issuer"],
        claims,
        expires: DateTime.Now.AddMinutes(120),
        signingCredentials: credentials);

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Figure 64 JWT generation

Password encryption

A hash is a one-way function, so given the password, you can work out the hash, but given the hash, you can't get the original password back [12]. When encrypting passwords for our users, we used the IPasswordHasher<TUser> interface, the default hashing and salting interface for .NET core applications, which has the following specification:

- **PBKDF2 with HMAC-SHA256**
    - **PBKDF2** is a simple cryptographic key derivation function, which is resistant to dictionary attacks and rainbow table attacks.
    - **HMACSHA256** is a type of keyed hash algorithm that is constructed from the SHA-256 hash function and used as a Hash-based Message Authentication Code (HMAC). The HMAC process mixes a secret key with the message data, hashes the result with the hash function, mixes that hash value with the secret key again, and then applies the hash function a second time. The output hash is 256 bits in length [13].

- **128-bit salt**
- **256-bit subkey**
- **10000 iterations**
    - The number of iterations the HMAC should be applied

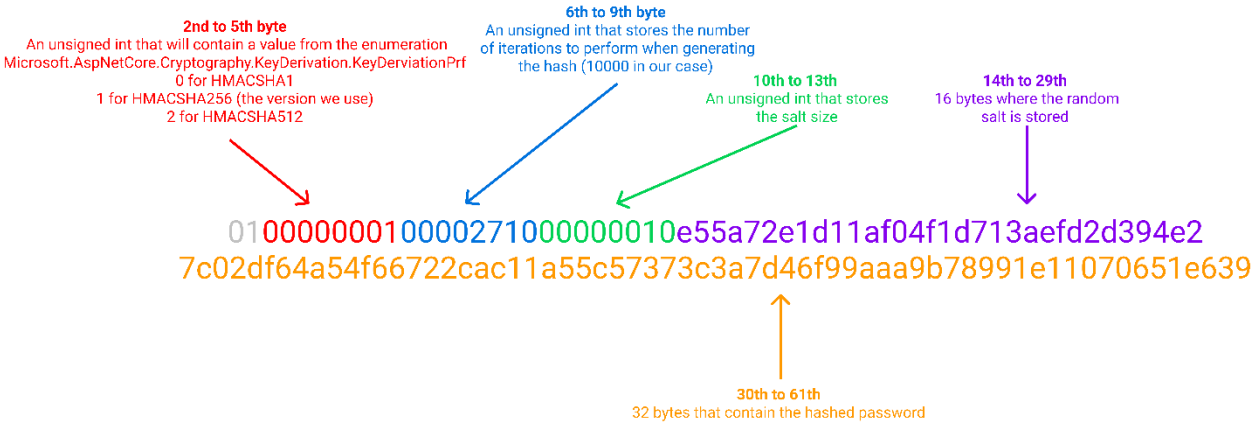

Figure 15 - The Anatomy Of Our Hash

5. Security

5.1. 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. The aftermath of this vulnerability can result in 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.

```
public Assignment GetByAssignmentId(int assignmentId)
{
    try
    {
        return _db.QueryFirst<Assignment>("Select * from [dbo].[Assignment] where assignmentId=@assignmentId", new { assignmentId = assignmentId });
    }
    catch (SqlException e)
    {
        throw e;
    }
}
```

Figure 16 Parametrized dapper query

5.2. 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 website's database and the scripts could then be executed on the original website without the user's knowledge. The key to taking 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.

```
@model Models.Assignment;
@{
    ViewData["Title"] = "Create Assignment";
    Layout = "~/Views/Shared/_LayoutWithUserSidebar.cshtml";
}

@section Stylesheets {
    <link rel="stylesheet" href="/css/create-assignment.css" />
}

<main>
    <div class="container-fluid">
        <form asp-controller="Assignment" enctype="multipart/form-data" asp-action="CreateAssignment" method="post">
            @Html.AntiForgeryToken()

            <div class="row">
                <div class="col-md-12 screen-1300-full">
                    <div class="boxed-row">

                        <div class="row create-assignment-title-block">
                            <div class="col-md-9">
                                <h3 class="create-assignment-page-title">
                                    Create Assignment
                                </h3>
                                <span class="@ViewBag.ResponseStyleClass">@ViewBag.Message</span>
                            </div>
```

Figure 17 HTML encoding

5.3. **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 in 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.

```
/*can be accessed by everybody who
 * is logged in
 */
[Route("assignment/create-assignment")]
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> CreateAssignmentAsync(IFormCollection collection, IFormFile files)
{
    try
    {
```

Figure 78 Method with ValidateAntiForgeryToken

5.4. 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 automated software that executes the task. Depending on the strength of your passwords these attacks can take anywhere from seconds to many years. [14] This is the reason why it is important to use strong passwords and encryption. Our minimal password requirements include at least six characters, an upper case character, a lower case character, a numeric value, and a 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.

5.5. 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 of requests to a website, 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 to the website. [15]

5.6. 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 (e.g. 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. To prevent this, we would configure our servers to use HTTPS.

5.7. **Over Posting attack**

Overposting, also known as mass assignment, is a type of cyberattack where the attacker is exploiting a 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 can set values to properties, he normally would not be able to. [16] 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.

6. Concurrency

Concurrency generally is the program's 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 take care of.

6.1. 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 afterward. It does not limit the users in any way, which means it is the best solution for systems with a 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.

6.2. 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. [17]

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 (*Figure 19/20*).

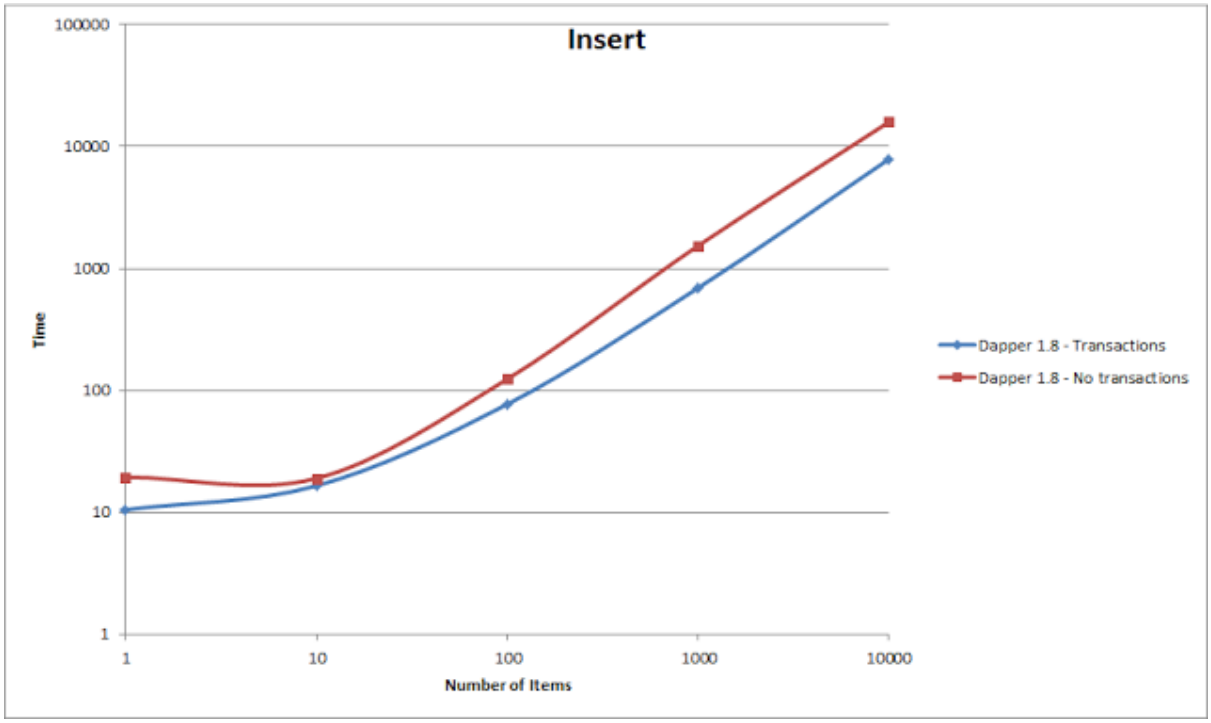

Figure 19 Insert comparison

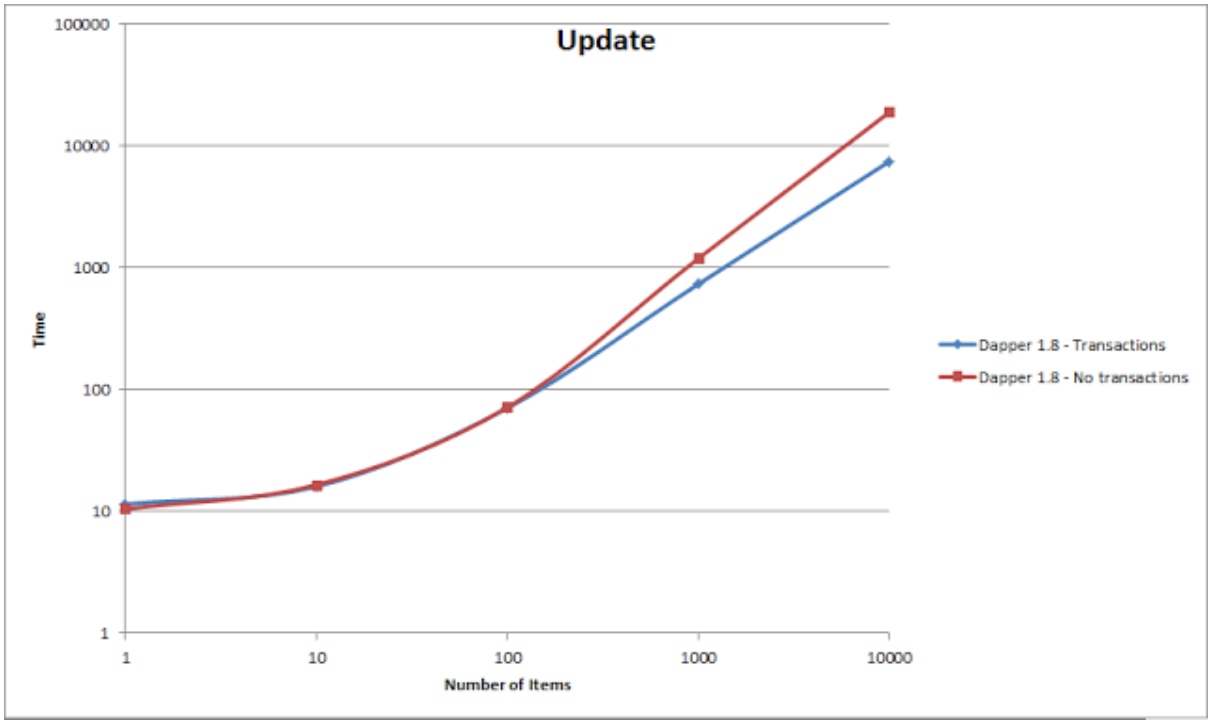

Figure 80 Update comparison

It must be noted that naturally, each transaction has an overhead, meaning the fewer 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 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.

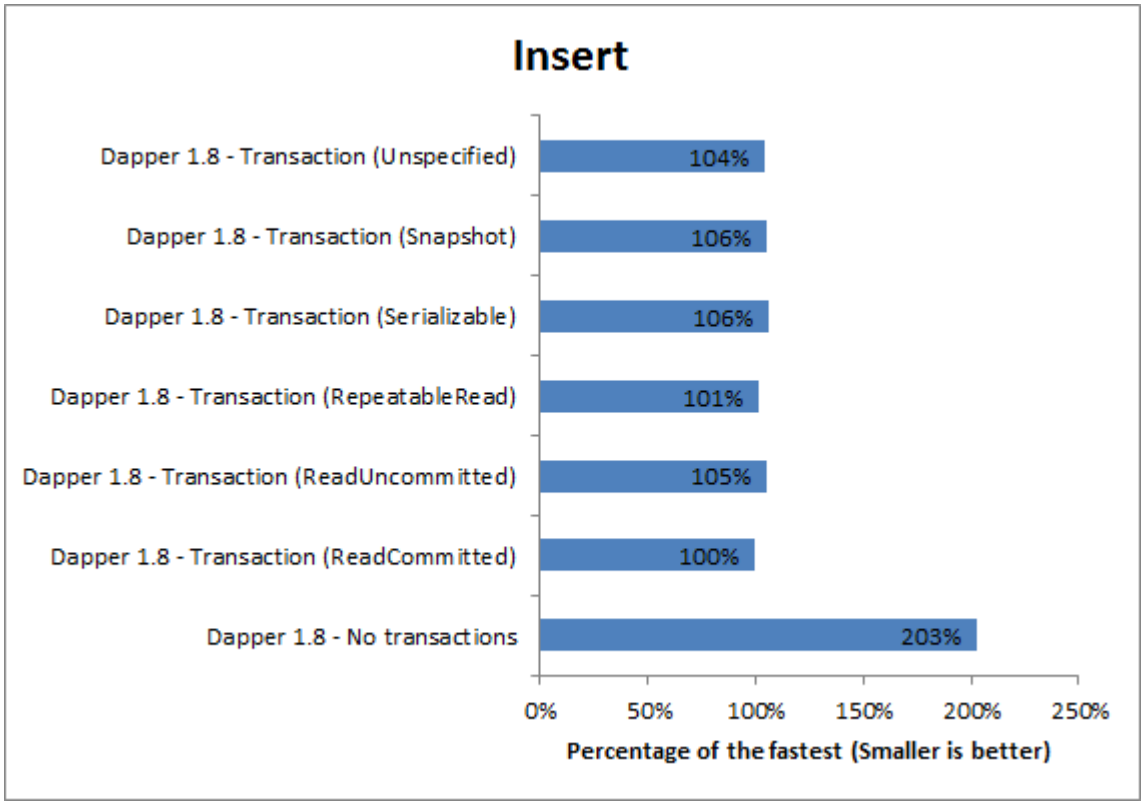

Figure 91 Isolation levels comparison

6.3. Our concurrency

Let us look at how different multi-user issues were solved in our system. For the majority of the issues, we chose to use 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 the user confirms the solution to an assignment, he locks the database for a moment (*Figure 22)*. 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.

```
12 references
public int CreateSolution(Solution solution)
{
    try
    {
        int lastUsedId;
        //this closing of the dbConnection is due to tests
        _db.Close();
        _db.Open();
        using (var transaction = _db.BeginTransaction(IsolationLevel.Serializable))
        {
            try
            {
                //try if creation was successful
                lastUsedId = _db.ExecuteScalar<int>(
                                            @"INSERT INTO [dbo].[Solution](assignmentId, userId, description, timestamp, solutionRating, anonymous, accepted)
                                            OUTPUT INSERTED.solutionId " +
                                            "VALUES (@assignmentId, @userId, @description, @timestamp, @solutionRating, @anonymous, 0)",
                                            new
                                            {
                                                assignmentId = solution.AssignmentId,
                                                userId = solution.UserId,
                                                description = solution.Description,
                                                timestamp = solution.Timestamp,
                                                solutionRating = solution.SolutionRating,
                                                anonymous = solution.Anonymous
                                            }, transaction: transaction);

                if (solution.SolutionFile != null)
                {
                    _db.Execute(@"INSERT INTO [dbo].[SolutionFile](solutionId, solutionFile) values (@solutionId, @solutionFile)",
                        new { solutionId = lastUsedId, solutionFile = solution.SolutionFile }, transaction: transaction);
                }
```

Figure 102 Database lock

```
            if (solution.SolutionFile != null)
            {
                _db.Execute(@"INSERT INTO [dbo].[SolutionFile](solutionId, solutionFile) values (@solutionId, @solutionFile)",
                    new { solutionId = lastUsedId, solutionFile = solution.SolutionFile }, transaction: transaction);
            }

            if (lastUsedId > 0)
            {
                int noOfSolutions = _db.QueryFirst<int>("SELECT COUNT(*) FROM [dbo].[Solution] where assignmentId=@assignmentId",
                    new { assignmentId = solution.AssignmentId }, transaction: transaction);
                //check if solution is the first in the queue
                if (noOfSolutions == 1)
                {
                    transaction.Commit();
                    _db.Close();
                    return 1;
                }

                if (noOfSolutions>1)
                {
                    int lastPostedSolutionId = _db.QueryFirst<int>("SELECT [solutionId] FROM [dbo].[Solution] where assignmentId=@assignmentId " +
                        "order by timestamp DESC", new { assignmentId = solution.AssignmentId }, transaction: transaction);
                    //checks if the Id put into the DB really is last
                    if (lastPostedSolutionId == lastUsedId)
                    {
                        bool isAssignmentActive = _db.QueryFirst<bool>("Select [isActive] from [dbo].[Assignment] where assignmentId=@assignmentId",
                            new { assignmentId = solution.AssignmentId }, transaction: transaction);
                        //check if assignment of the solution is active
                        if (isAssignmentActive)
                        {
                            transaction.Commit();
                            _db.Close();
                            return noOfSolutions;
                        }
                    }
                }
            }
            transaction.Rollback();
```

Figure 113 Create Solution method

The 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 the database (*Figure 24*). 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.

```
4 references
public int UpdateAssignment(Assignment assignment, int assignmentId)
{
    try
    {
        byte[] inputTimestamp = assignment.Timestamp;
        int returni =  _db.Execute(@"Update [dbo].[Assignment] set title=@title,
            description=@description, price=@price, deadline=@deadline, anonymous=@anonymous, academicLevel=@academicLevel, subject=@subject
            WHERE assignmentId = @assignmentId AND timestamp = @timestamp",
            new { title = assignment.Title, assignmentId = assignmentId, description = assignment.Description, price = assignment.Price,
                deadline = assignment.Deadline, anonymous = assignment.Anonymous, academicLevel = assignment.AcademicLevel,
                subject = assignment.Subject, timestamp=inputTimestamp });
        return returni;
    }
    catch (SqlException e)
    {
        throw e;
    }
}
```

Figure 124 Update assignment method

The user is adding credits to his account and at the same time moderator accepts the credit return request and adds the credits to the 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 (*Figure 25*). 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.

```
3 references
public int IncreaseUserCredits(int credits, string userId)
{
    try
    {
        User concurrencyInfo = _dbUser.GetUserCredits(userId);
        return _dbUser.UpdateUserCredits(credits + (int)concurrencyInfo.Credit, userId, concurrencyInfo.ConcurrencyStamp);
    }
    catch (Exception e)
    {
        throw e;
    }
}
```

```
public int UpdateUserCredits(int credit, string userId, string concurrencyStamp)
{

    _db.Open();
    using (var transaction = _db.BeginTransaction(IsolationLevel.Serializable))
    {
        try
        {

            int updateResult = _db.Execute(@"Update [Identity].[User] set credit=@credit
                WHERE Id = @userId AND ConcurrencyStamp = @ConcurrencyStamp",
                new { credit = credit, userId = userId, ConcurrencyStamp = concurrencyStamp }, transaction: transaction);
            if (updateResult > 0)
            {
                string newGuid = Guid.NewGuid().ToString();
                int generated = _db.Execute(@"Update [Identity].[User] set ConcurrencyStamp=@concurrencystamp
                    WHERE Id = @userId",
                    new { userId = userId, concurrencystamp = newGuid }, transaction: transaction);
                if (generated > 0)
                {
                    transaction.Commit();
                    _db.Close();
                    return updateResult;

                }
            }
            transaction.Rollback();
            _db.Close();
            return -1;

        }
```

Figure 135 Update credits method

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 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 (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 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.

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 could occur when the poster goes through all the solutions, decides to choose one, and pays 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.

Not implemented concurrency problems are described in *Appendix E.*

7. Testing

In the agile methodology, testing is done slightly differently than in plan-driven methodologies. The testing is a continuous process that is happening alongside, sometimes even before, development. Also, there are no specified testers, the code is being tested by developers themselves. In the end, what and how is tested is not based on a document with all the requirements, but on the acceptance criteria agreed upon between the customer and the development team. [18]

There are two main kinds of tests done in our project: unit testing and acceptance testing.

7.1. 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 starts 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 classes 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 afterward, so that someone else can pick up the code and easily work with it.

7.2. **Acceptance testing**

We wrote an acceptance test for each user story, to evaluate if it is finished (*Figure 26*). Usually in agile, they are written and performed by a customer. Since we do not have one, they were written by the whole team and performed by the developer or group of developers who implemented them.

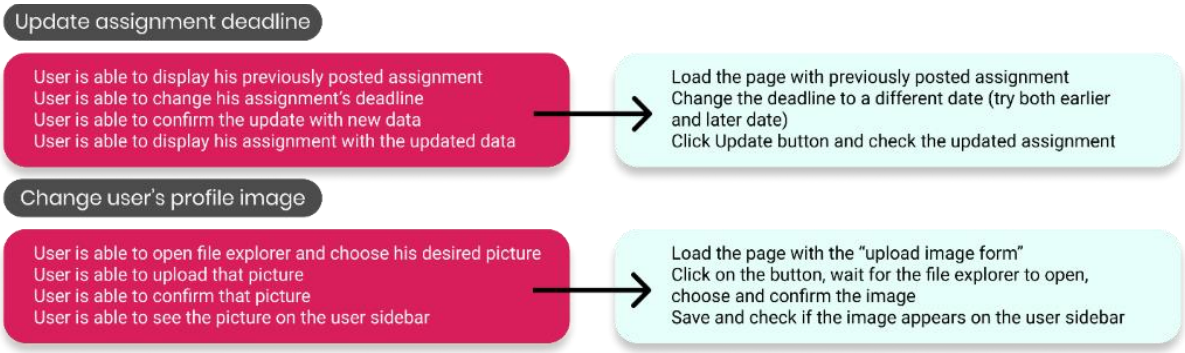

Figure 146 Acceptance test

8. Implementation

8.1. Interesting code snippets

First admin account

One question every development team must ask itself is how the first admin user is created. We have solved this in a convenient way in which the main program checks whether the admin account is in the database and if it is not - it creates it. That means that no matter what, every single time the program is run, we have an admin account to make changes a normal user would not be able to do.

```
0 references
public static async Task SeedSuperAdminAsync(UserManager<User> userManager, RoleManager<IdentityRole> roleManager)
{
    //Seed Default User
    var defaultUser = new User
    {
        UserName = "superadmin",
        Email = "we@thebest.com",
        FirstName = "We The",
        LastName = "Boys",
        EmailConfirmed = true,
        PhoneNumberConfirmed = true
    };
    if (userManager.Users.All(u => u.Id != defaultUser.Id))
    {
        var user = await userManager.FindByEmailAsync(defaultUser.Email);
        if (user == null)
        {
            await userManager.CreateAsync(defaultUser, "sebaHolesz1+");
            await userManager.AddToRoleAsync(defaultUser, Roles.Customer.ToString());
            await userManager.AddToRoleAsync(defaultUser, Roles.Moderator.ToString());
            await userManager.AddToRoleAsync(defaultUser, Roles.Admin.ToString());
        }

    }
}
```

Figure 27 First admin account

The assignment-user relationship

One important part of the system is how to realize who is the person looking at the assignment. Is he its author, someone who posted a solution to it, or an ordinary customer? We solve this in the API of the assignment.

```
[Route("check-user-vs-assignment/{assignmentId}")]
[HttpGet]
0 references
public IActionResult CheckUserVsAssignment(int assignmentId)
{
    try
    {
        string userId = APIAuthenticationController.GetUserIdFromRequestHeader(Request.Headers);
        int returnCode = AssignmentBusiness.GetAssignmentBusiness().CheckUserVsAssignment(assignmentId, userId);

        if (new[] { 0, 1, 2 }.Contains(returnCode))
        {
            return Ok(returnCode);
        }
        else
        {
            return NotFound("This user is not associated with the assignment!");
        }
    }
    catch (Exception)
    {
        return StatusCode(500);
    }
}

[Route("assignment/display-assignment/{assignmentId}")]
[HttpGet]
0 references
public ActionResult DisplayAssignment(int assignmentId)
{
    try
    {
        using (HttpClient client = new HttpClient())
        {
            User user = _userManager.FindByIdAsync(User.FindFirstValue(ClaimTypes.NameIdentifier)).Result;
            client.DefaultRequestHeaders.Authorization = AuthenticationController.GetAuthorizationHeaderAsync(_userManager, _signInManager, user).Result;

            string urlCheckUser = $"https://localhost:44316/apiV1/check-user-vs-assignment/{assignmentId}";
            HttpResponseMessage returnCodeRM = client.GetAsync(urlCheckUser).Result;

            if (returnCodeRM.IsSuccessStatusCode)
            {
                int returnCode = returnCodeRM.Content.ReadAsAsync<int>().Result;

                /*
                 * 0 = hes neither author nor previous solver
                 * 1 = authorUserId = currentUserId
                 * 2 = solverId = currentUserId
                 */
                switch (returnCode)
                {
                    case 0:
                        string urlCompleteAssignmentData = "https://localhost:44316/apiV1/assignment/complete-data/" + assignmentId;
                        HttpResponseMessage asuRM = client.GetAsync(urlCompleteAssignmentData).Result;
                        if (asuRM.IsSuccessStatusCode)
                        {
                            AssignmentSolutionUser asu = asuRM.Content.ReadAsAsync<AssignmentSolutionUser>().Result;
                            ViewBag.Assignment = asu.Assignment;
                            ViewBag.User = asu.User;

                            string urlCountOfSolutions = "https://localhost:44316/apiV1/solution/count-by-assignmentId/" + assignmentId;
                            HttpResponseMessage solutionCountRM = client.GetAsync(urlCountOfSolutions).Result;
```

Figure 28 Assignment-user relationship

Storing files included in assignments and solutions

For us, the biggest challenge was working with files. None of us had any previous experience with it, so we thought it would be a big accomplishment for us to make this feature work. In the end, we managed to make it work for assignments and solutions. We implemented it through form collection and here is a code snippet of how we did it.

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> CreateAssignmentAsync(IFormCollection collection, IFormFile files)
{
    try
    {
        using (HttpClient client = new HttpClient())
        {
            if (ModelState.IsValid)
            {
                User user = _userManager.FindByIdAsync(User.FindFirstValue(ClaimTypes.NameIdentifier)).Result;
                client.DefaultRequestHeaders.Authorization = AuthenticationController.GetAuthorizationHeaderAsync(_userManager, _signInManager, user).Result;

                Assignment assignment = new Assignment();
                assignment.Title = collection["Title"];
                assignment.Description = collection["Description"];
                assignment.Price = Convert.ToInt32(collection["Price"]);
                assignment.Deadline = Convert.ToDateTime(collection["Deadline"]);
                assignment.Anonymous = Convert.ToBoolean(collection["Anonymous"][0]);
                assignment.AcademicLevel = collection["AcademicLevel"];
                assignment.Subject = collection["Subject"];
                assignment.UserId = user.Id;

                if (files != null)
                {
                    var dataStream = new MemoryStream();
                    await files.CopyToAsync(dataStream);
                    assignment.AssignmentFile = dataStream.ToArray();
                    dataStream.Close();
                }
```

Figure 29 Storing file

User profile pictures

Another file we store in our database is the profile picture. The user can change the default picture we assign him by clicking on his name or by clicking ‘settings’ in the ever-present user sidebar. He then uploads the desired picture. If we had more time, we would include a check which would restrict the size of the upload.

```
<div class="col-md-6">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Input.ProfilePicture" style="width: 100%;"></label>
        @if (Model.Input.ProfilePicture != null)
        {
            <img id="profilePicture" style="width:350px;height:350px; object-fit:cover" src="data:image/*;
                base64,@(Convert.ToBase64String(Model.Input.ProfilePicture))">
        }
        else
        {
            <img id="profilePicture" style="width:350px;height:350px; object-fit:cover" src="">
        }
        <input type="file"
               accept=".png,.jpg,.jpeg,.gif,.tif"
               asp-for="Input.ProfilePicture"
               class="form-control"
               style="border:0px!important;padding: 0px;padding-top: 10px;padding-bottom: 30px;"
               onchange="document.getElementById('profilePicture').src = window.URL.createObjectURL(this.files[0])" />
        <span asp-validation-for="Input.ProfilePicture" class="text-danger"></span>
    </div>
</div>
```

Figure 30 Profile picutre

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.

```
[AllowAnonymous]
[Route("assignment/display-assignments-page/{pageNumber}")]
[HttpGet]
0 references
public ActionResult DisplayAllAssignments(int pageNumber)
{
    try
    {
        using (HttpClient client = new HttpClient())
        {
            if (User.Identity.IsAuthenticated)
            {
                User user = _userManager.FindByIdAsync(User.FindFirstValue(ClaimTypes.NameIdentifier)).Result;
                client.DefaultRequestHeaders.Authorization = AuthenticationController.GetAuthorizationHeaderAsync(_userManager, _signInManager, user).Result;

                string urlGetAllAssignments = $"https://localhost:44316/apiV1/assignment/page-all-active-not-posted-by-user/{pageNumber}";

                HttpResponseMessage assignmentsNotPostedByUserRM = client.GetAsync(urlGetAllAssignments).Result;
                if (assignmentsNotPostedByUserRM.IsSuccessStatusCode)
                {
                    ViewBag.Assignments = assignmentsNotPostedByUserRM.Content.ReadAsAsync<List<Assignment>>().Result;
                    Page page = JsonConvert.DeserializeObject<Page>(assignmentsNotPostedByUserRM.Headers.GetValues("PagingHeaders").FirstOrDefault());
                    ViewBag.NextPage = pageNumber + 1;
                    ViewBag.PreviousPage = pageNumber - 1;
                    ViewBag.Link = "/assignment/display-assignments-page/";
                    ViewBag.PreviousEnable = page.PreviousPage == true ? "" : "disabled";
                    ViewBag.NextEnable = page.NextPage == true ? "" : "disabled";
                    ViewBag.PageNumber = pageNumber;
                    ViewBag.TotalPages = page.TotalPages;
                    return View("AllAssignments");
                }
```

Figure 31 Pagination

CONCLUSION

To conclude our 3rd-semester project, during which we were developing an MVP for our new C2C platform, Solvr.Online, we need to look back and reflect on the process. During the past weeks, we have gained experience in creating applications as distributed systems, using MVC for web client and WPF for desktop client. We considered different security vulnerabilities and dealt with concurrency issues when 2 different users try to do the same request. We have worked in 3-tier architecture which ensured good separation of data, business logic, and UI. For accessing the database we have used ORM – dapper and tried both code-first and database-first approach. At the end of the project, we were satisfied that we finished what we planned, but there is still space for improvements which we keep in mind.

REFERENCES

[1] "Dapper Tutorial," [Online]. Available: https://dapper-tutorial.net/dapper.

[2] Lithmee, "What is the Difference Between Code First and Database First Approach in MVC," 22 6 2019. [Online]. Available: https://pediaa.com/what-is-the-difference-between-code-first-and-database-first-approach-in-mvc/.

[3] "SOAP vs. REST: A Look at Two Different API Styles," Upwork , 19 4 2017. [Online]. Available: https://www.upwork.com/resources/soap-vs-rest-a-look-at-two-different-api-styles.

[4] M. DUBOSE, "WPF Vs WinForms – What To Choose?," 4 6 2019. [Online]. Available: https://www.rdglobalinc.com/wpf-vs-winforms-what-to-choose/.

[5] lrshivangini, "Security Boulevard - Authentication vs. Authorization Defined: What's the Difference?," [Online]. Available: https://securityboulevard.com/2020/06/authentication-vs-authorization-defined-whats-the-difference-infographic/.

[6] A. Ahmed, "Better Programming - JSON Web Tokens vs. Session Cookies for Authentication," 2020. [Online]. Available: https://medium.com/better-programming/json-web-tokens-vs-session-cookies-for-authentication-55a5ddafb435.

[7] Auth0, "JWT.IO - Introduction To JSON Web Tokens," [Online]. Available: https://jwt.io/introduction/.

[8] Hackernoon, "Hackernoon - Using Session Cookies Vs. JWT for Authentication," 2020. [Online]. Available: https://hackernoon.com/using-session-cookies-vs-jwt-for-authentication-sd2v3vci.

[9] F. Copes, "LogRocket - JWT authentication: When and how to use it," 2018. [Online]. Available: https://blog.logrocket.com/jwt-authentication-best-practices.

[10] "JWT.IO - Introduction to JSON Web Tokens," [Online]. Available: https://jwt.io/introduction/.

[11] "SSL2BUY - Symmetric vs. Asymmetric Encryption – What are differences?," [Online]. Available: https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences.

[12] A. Lock, "Andrew Lock - Exploring the ASP.NET Core Identity PasswordHasher," 2017. [Online]. Available: https://andrewlock.net/exploring-the-asp-net-core-identity-passwordhasher/.

[13] Microsoft, "HMACSHA256 Class," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.hmacsha256?view=net-5.0.

[14] "Brute Force Attack: Definition and Examples," Kaspersky, [Online]. Available: https://www.kaspersky.com/resource-center/definitions/brute-force-attack.

[15] "Dynamic IP Restrictions," Microsoft, [Online]. Available: https://www.iis.net/downloads/microsoft/dynamic-ip-restrictions.

[16] "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/.

[17] "Making Dapper Faster with Transactions," 26 4 2012. [Online]. Available: https://blog.staticvoid.co.nz/2012/making_dapper_faster_with_transactions/.

[18] "Understanding Agile Testing Methodology and 4 Agile Testing Methods," [Online]. Available: https://www.sealights.io/agile-testing/understanding-agile-testing-methodology-and-4-agile-testing-methods/.

[19] "GeeksForGeeks," 14 8 2020. [Online]. Available: https://www.geeksforgeeks.org/sql-transactions/.

[20] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Deadlock.

[21] "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 B – Relational Model

The second step we focused on after the domain model was mapping it into the Relational Model (*Figure 32*). 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 the User into two columns – firstName and lastName to achieve atomicity. Also, we applied the Id column as a primary key of most relations, so all other values depend on the whole key – 2nd normal form.

This was the first version of it, which includes all tables, even those we did not have time to implement. During the process, it was also edited due to implementing Identity for Authentication & Authorization, which replaced our User relation.

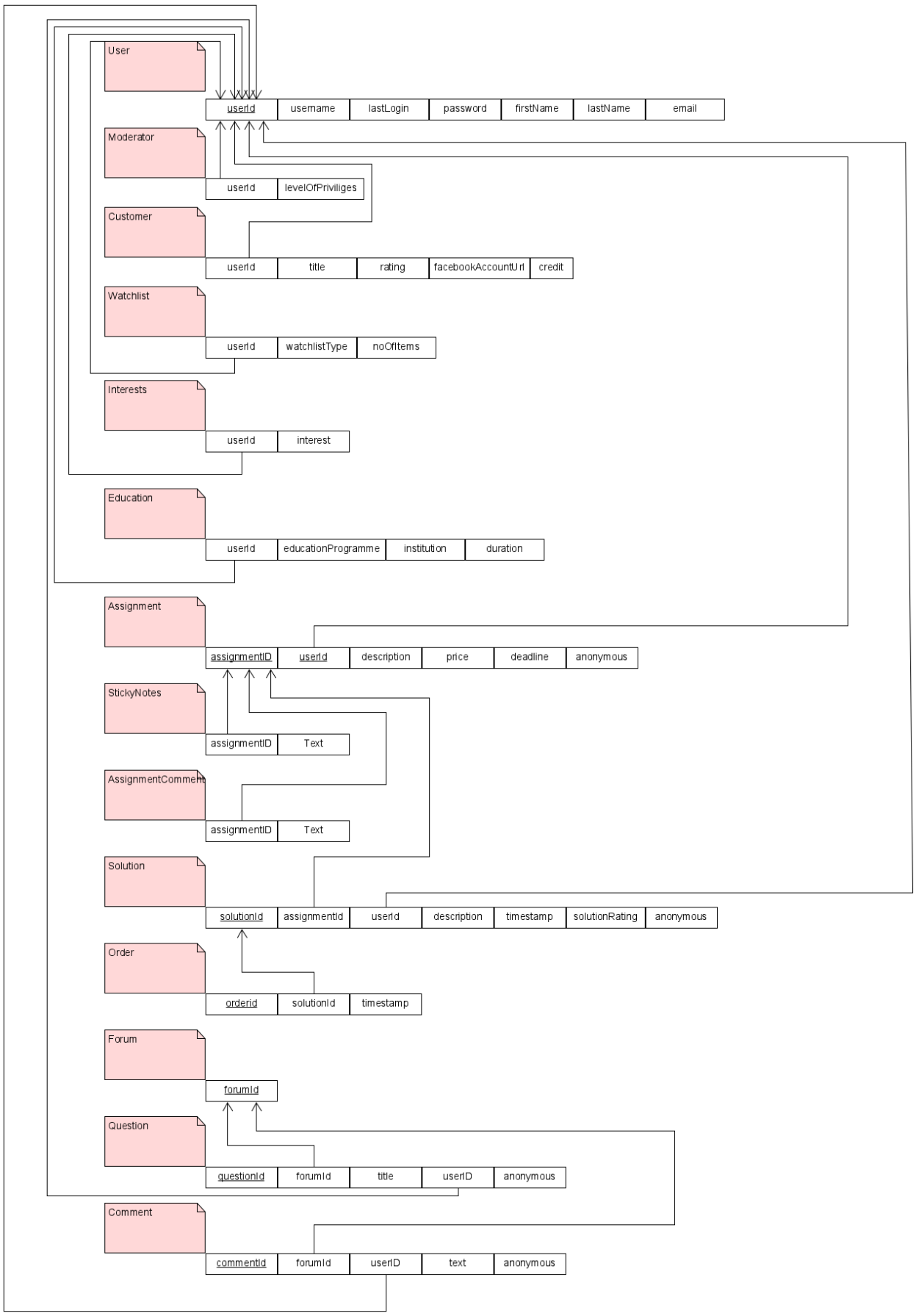

Figure 32 Relational Model

Appendix C - 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 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, First-Name)
    - 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 an underscore (e.g. _connection)
2. Declaration
    - Declaring all class members at the top of the class
    - Declaring local variables close to their usage
3. 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
4. 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 that have different purposes
- We avoided type identification in the naming of variables and methods. (e.g. strValue)

Appendix D

1. User's browser sends an HTTP Get request to the MVC web application to display the View with the Login form
2. a) User's browser sends an HTTP Post request to the MVC web application containing his username/email and password, together with an AntiForgeryToken to avoid CSRF

   b) The ASP.NET Identity service retrieves the user's password hash from the Identity DB, checks if the saved password hash matches a hash of the incoming password.

   c) If the login attempt is successful, the MVC web application makes an HTTP Post request to the web API containing the user's username/email and password, returning a new JWT if successful. This JWT will be used as an authorization token for that specific user.

   d) The JWT is inserted into the Identity DB. After all the previous steps go through, the Cookie and JWT are saved, the user is redirected to the home page.
3. a) User's browser sends an HTTP Get request to the MVC web application to display the View with the Update Assignment form. The system checks for the Cookie to authorize the user.

   b) The MVC web application makes an HTTP Get request to the web API containing the user's JWT as a part of the authorization header to check the user's relation to the specific assignment. This HTTP request is to make sure that a user does not access a View he is not authorized to. If the user who is not an author of that assignment tries to access the Update Assignment page, he is redirected to the Display Assignment page instead. The internal return code 1 means that the user is also an author of the assignment.

   c) After the MVC web application makes sure the user is authorized to access the Update Assignment page, it sends an HTTP Get request to the web API to retrieve the assignment data, so it can prefill the Update Assignment View with it. The request authorization header again contains the JWT to authorize the user.

   d) The API selects the data from the DB and sends it back to the MVC web application, which renders the correct View and displays it to the user.
4. a) User changes the desired data and confirms the update. Then the user's browser sends an HTTP Put request containing the form data to the MVC web application. The system again checks the Cookie to authorize the user.

b) The MVC web application takes the form data from the FormCollection object and creates an assignment object. Then it makes an HTTP Put request to the web API containing the assignment object and the JWT as a part of the authorization header. The web API and the business layer beneath it check if the request is authorized and updates the assignment.

c) The updated assignment is put into the DB. After a successful update, the web API and then the MVC web application returns a success message as a part of the Success View.

Appendix E – Concurrency

Moderator deleting someone's post for 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 the same solution we use for the problem where a 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 his 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 that would enable the moderator to ban users.