3rd Semester Project Report

PROGRAMMING & TECHNOLOGY

Project Participants:

- **Dominik Ábel Sári** <1081511@ucn.dk>
- **Erik Petra** <1081490@ucn.dk>
- **Gyozo Csuhai** <1081502@ucn.dk>
- **Krisztián Henrik Papp** <1081477@ucn.dk>
- **Marek Strúcka** <1081474@ucn.dk>

Supervisors:

- **Nadeem Iftikhar, Ph.D.** <naif@ucn.dk>
- **Michael Holm Andersen** <mihn@ucn.dk>

Repository path:
https://github.com/dmai0919-group3/3rd-semester-project

Normal pages / characters: 19.9 pages / 47 819 characters*

*A normal page is defined as 2400 characters incl. whitespaces and footnotes. Frontpage, title page, table of contents, literature list and appendices are not included.

DEC 21, 2020

University College of Northern Denmark
AP Degree in Computer Science
DMAI0919 – Group 3

PROFESSIONSHØJSKOLEN

Table of Contents

Abstract

This report describes the implementation of a tool for handling files among users and user groups from a technological point of view; reader could read about the coding and technological decisions that were undertaken to complete this project; discussing the overall design and architecture, continuing through a section aimed at choosing and implementing a database, discussion about services and security, finishing it off by the description of concurrency problems.

Preface

We would like to thank all our teachers for their invaluable help, guidance and feedbacks that we received during the whole duration of the project. They always patiently answered our questions, came to meet us and help us with a supportive and caring attitude and have therefore helped us a lot to finish this project.

Introduction

The increasing trend in usage of file sharing and file storage applications on the cloud platforms all around the world is undisputable. According to reports, Google Suite has more than 1 billion active users, 1000s of organizations that use their Drive and millions of users that pay for this service. [1]

The idea of having all the files needed for work located rather on cloud, where it is backed up and accessible for everyone with permission, than on a local storage somewhere, is appealing – in fact, 90% of companies nowadays use cloud services running 60% of their workloads there. [2] When combined with its flexibility, disaster recovery and the ability to ease the work, the upsurge of cloud in the upcoming years is inevitable.

That is why the group decided to try to create a file sharing system that will use cloud services, where people would be able to register and log in, have both their own private files and shared files among groups they will create together with a lot of additional smaller features.

Problem Statement

The problem

All creative workflows, regardless of the field, technical aspects or level of expertise, rely on communication and collaboration between individuals and groups of people. The issue is that there is no easy-to-use group collaboration software aimed at file sharing.

Our aim is to provide a file-sharing system offering easy and efficient sharing, versioning and commenting of documents and other files, to ease collaborative efforts within projects involving multiple persons.

Problem statement

Concerning programming and technology in our project, we identified four important questions in our complete problem statement (to be seen at Appendix X) related to this:

- Are we able to version the files and to store their previous versions, so that the users could access them easily?
- Are we able to prevent the user from losing the progress he made while editing or commenting on a document by either caching or autosaving?
- Are we able to resolve the concurrency issues (two users working on the same file at the same time) in the system?
- Will the data in our system be safe and protected against various attacks?

We strive to complete the project to the best possible extent and to provide answers to these questions once the project is finished.

Project Configuration

Scrum combined with XP practices will be the main development approach used to complete this project. We plan to use the .NET Core Framework as our main development technology. We plan to structure our application, so what it will have two clients (an ASP.NET MVC web client and a WPF desktop client) which will connect and get information from a RESTful API service (which will be based on ASP.NET Web API).

For version control, we are going to use Git and host our repositories on GitHub. For the database, we are going to use a Microsoft SQL Server, hosted and managed by Azure in the form of an Azure SQL Database. We are going to deploy the finished web app (including the web client and API) to Azure App Service under a custom domain name.

Structure of the report

This report is divided into X distinct parts that describe various considerations and solutions that were discussed and implemented in this project. The report started with an abstract, a preface and an introduction after which a reader could expect to see a detailed report setup together with additional helpful information. This is followed by these parts:

- Part I. ***Design and Architecture***
- Part II. ***Technologies Used***
- Part IV. ***Security***
- Part V. ***Patterns***
- Part VI. ***Other Interesting Details***
- Part VIII. ***Conclusion***

The report ends with an Appendices part, where things like a full problem statement or a group contract can be found.

Every class or method name mentioned in the report is formatted as such: `Class.Method()`

Helpful information

GitHub Repository path: https://github.com/dmai0919-group3/3rd-semester-project
Deployed Web App at Azure App Service: https://ogo-file.space/user/login
Final executable release of the Desktop Client: https://github.com/dmai0919-group3/3rd-semester-project/releases/tag/v1.0.0

Part I. Design & Architecture

1. Architectural choices

There are plenty architectural patterns to choose from that ease the process. The most common one is layered architecture with separation of concerns into layers on the same computer, then there is N-tier architecture which resembles the layered one, but now the “tiers” are on distinct computers, then there is Client/Server architecture, where client communicates with the servers via requests.

1.1 Architecture in our project

In the beginning we had to choose between .NET Framework and .NET Core. Initially we wanted to go with .NET as that is what we used during the semester at Nadeem’s classes, however during a spike in Sprint 0, we have found out that the Azure library we wanted to use heavily depends on some asynchronous functions that were introduced in .NET Core 3.1 and are not included in the current release of .NET 4.7. Another thing that influenced our decision is that Microsoft’s focus currently is on .NET Core as in the upcoming .NET Version 6 they plan to “merge” the two together to have one stable, cross-platform framework instead of the two separate ones, which will be based on .NET Core rather than the normal .NET Framework.

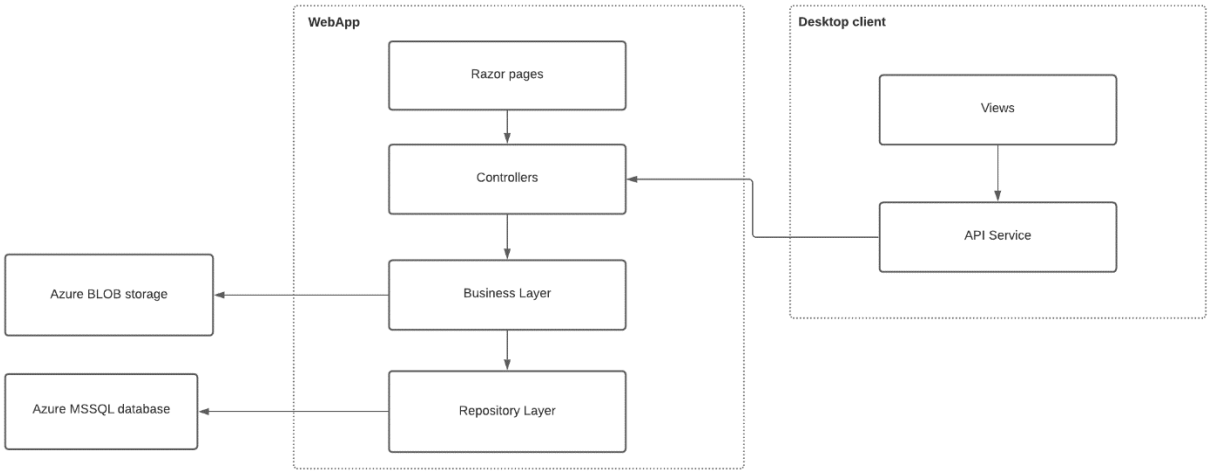


Figure 1: Visualized architecture

We structured our project into two separate Visual Studio Projects inside a single Solution. Reasoning for it is explained more in-depth in the part about Web app.

2. Brief description of layers

2.1 Web Client and Web API

2.1.1 Repository Layer

The repository layer that we implemented is only used to communicate directly with the database. It contains services and methods specifically created for our needs. It mainly uses the micro ORM Dapper to ease mapping of database results into our models and entities. It also allows us to write custom SQL scripts.

2.1.2 Business Logic Layer

We introduced the business logic layer as we started working on the UserController and UserApiController classes. We saw a lot of code that was doing the same thing, so we decided to introduce business logic layer, at first containing UserService.

UserService contains all the logic that handles login, register, password hashing and other user related important functionalities we need. It also directly communicates with the repository layer.

Business logic also handles access logic (authorization as in access to files and not authorization). Example of that can be seen in FileService, where in methods that handle file operations, we check whether logged in user has permission to work with that file.

2.1.3 Controller Layer

2.1.3.1. API Controllers

API Controllers offer REST API endpoints. These contain most of the logic, since we are designing our web client to be using a lot of JavaScript and behave dynamically.

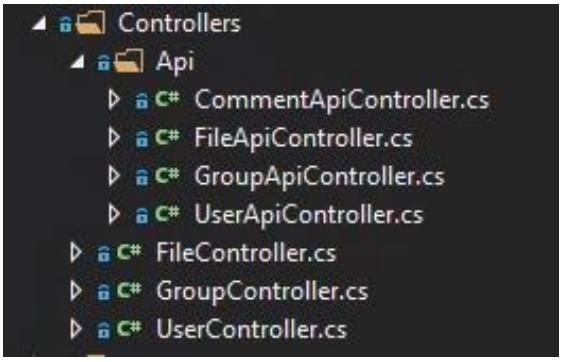


Figure 2: Folder containing all Controllers

2.1.3.2. MVC Controllers

MVC Controllers are used for displaying the views in our web application. Since we took the approach of using jQuery in most of the front-end, these controllers are small compared to their API counterparts. A great example of that is FileController.

```
namespace Group3.Semester3.WebApp.Controllers
{
    [Route("file")]
    [Authorize(AuthenticationSchemes = CookieAuthenticationDefaults.AuthenticationScheme)]
    1 reference | Erik, 4 days ago | 6 authors, 17 changes
    public class FileController : Controller
    {
        private IFileService _fileService;
        private IUserService _userService;

        0 references | Erik, 4 days ago | 1 author, 3 changes
        public FileController(IFileService fileService, IUserService userService)
        {
            _fileService = fileService;
            _userService = userService;
        }


        /// <summary> GET: file/browse
        [Route("browse")]
        [HttpGet]
        0 references | Krisztián Henrik Papp, 7 days ago | 4 authors, 5 changes
        public ActionResult Browse()...

        /// <summary> GET: file/shared/{hash}
        [Route("shared/{hash}")]
        [HttpGet]
        [AllowAnonymous]
        0 references | Erik, 4 days ago | 1 author, 1 change
        public ActionResult SharedFileLink(string hash)...

    }
}
```

Figure 3: Implementation of FileController

Our FileController has only two methods for accepting requests. On the other hand, FileApiController has over 500 lines of code with more than 15 methods. This was possible thanks to our approach when it comes to web front-end.

2.2 Desktop Client

2.2.1 Summary

We chose creating a modern client with a dynamic user experience as our primary objective. To achieve this, we used a multitude of technologies, such as the Windows Presentation Foundation (WPF) UI framework along with a material design toolkit [3], and a JSON framework [4] for communicating with the backend through the API.

During development, the web application served as a testbed to the features which would only later get implemented in the desktop client. This was due to the frequent changes in the business logic, which, when used by the desktop client would have required updating the web API, and the corresponding API connector service of the client.

2.2.2 WPF

The Windows Presentation Foundation (WPF) is a graphical UI framework used to create applications with a rich user experience, providing extensive styling and rendering options. WPF supports 2D and 3D graphics and utilizes a vector-based rendering engine, which can use hardware acceleration of modern computers, making the UI faster, scalable and resolution independent.

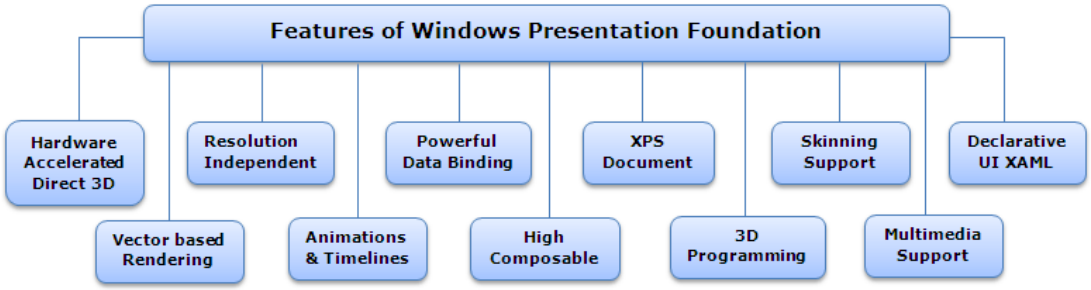


Figure 4: Features of the Windows Presentation Foundation [5]

WPF uses a markup language (Extended Application Markup Language or XAML) for defining UIs. These UI definitions are separate from the application code and interaction logic, allowing the programmer to rapidly test and develop UIs.

2.2.2.1. The Model-View-ViewModel Pattern

Another powerful feature of WPF is the support of the Model-View-ViewModel (MVVM) design pattern, developed specifically for the framework. It is a variation of the MVC pattern, tailored for use with the WPF and the Microsoft Silverlight framework.

MVVM aims to separate of development of the graphical user interface with the help of a GUI mark-up language, backed up with interaction logic code connecting the View to the ViewModel and handling GUI events.

Another key difference between MVC and MVVM is that in MVVM the entry point to the application is the View, not the controller (the interaction logic). The interaction code only serves the purpose of handling user interface interactions, while the UI itself is self-contained and is automatically built and updated from the ViewModel.

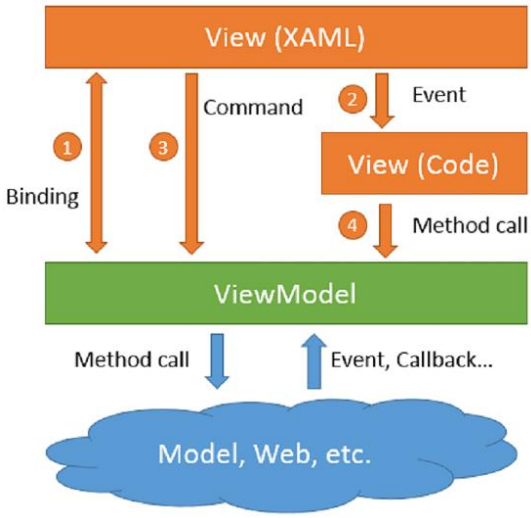


Figure 5: Data flow in MVVM

- In MVC, controller is the entry point to the Application, while in MVVM, the view is the entry point to the Application.
- MVC Model component can be tested separately from the user, while MVVM is easy for separate unit testing, and code is event driven.
- MVC architecture has "one to many" relationships between Controller & View while in MVVC architecture, "one to many" relationships between View & View Model. [17]

2.2.2.2. Data Binding

"Data binding is general technique that binds two data/information sources together and maintains synchronization of data." [6]

The hacky bit is supplying the correct property's name to the event handlers. This, as suggested by Microsoft [The main way of interacting with data within the WPF framework is data binding, a mechanism which allows modifications of the business model to be instantly reflected by the UI, and the UI to change the application state and update fields of the model without the need of extra interaction logic.

Bindings can be uni- or bidirectional. Bindings can be created in the View XAML by using the {Binding} mark-up extension.

```
<Grid DataContext="[...]">
    <TextBox Text="{Binding EmployeeNumber}"></TextBox>
    <TextBox Text="{Binding Name}"></TextBox>
    <TextBox Text="{Binding Title}"></TextBox>
</Grid>

public EmployeeModel {
  public string EmployeeNumber { get; set; }
  public string Name { get; set; }
  public string Title { get; set; }
}
```

Bindings are evaluated in real time, and work within their respective DataContexts. There is no default source for the DataContext property, and there are multiple ways to set the DataContext

property of an object. It can be an arbitrary object representing the model which the UI element utilizing the data would reflect. In the above example, let us assume that the Grid's DataContex was set to an instance of the EmployeeModel class. In this case, the fields would automatically get updated with the values of the corresponding TextBoxes' contents.

In our case, we also needed changes in the ViewModel to update the UI. Each side of a binding can be a normal .NET property (members which, to the outside, behave like variables, but consist of special methods for reading and writing their values, called accessors) or a DependencyProperty. With normal properties, which our models mainly consist of, for UI updates to work properly, models must implement the INotifyPropertyChanged interface. The GUI handler routine automatically signs up to the PropertyChanged event provided by the interface and updates the user interface when the event is fired.

The hacky bit is supplying the correct property's name to the event handlers. This, as suggested by Microsoft [7]. Another interesting scenario we encountered was the handling of popup windows, such as context menus, and material design dialogs. These are separate windows, but they do not share the DataContext of their parent element or window as they are not part of the view tree, even though they are defined in the same markup code and are properties of the UI elements in the tree. Therefore, it is problematic to access the data of elements which have a context menu belonging to them from the menu's code. To overcome this, we set a Tag on items which have a context menu using a RelativeSource Binding to find the ancestor Window of the items. As the items displayed are part of the view tree, the tag property gets set to the DataContext of the window, which we can access from the context menu by referencing the Tag of the menu's placement target, in this case the item of interest, and set it as the menu's DataContext. This way we can make the ContextMenu share the same DataContext as the item it belongs to.

```
</ListView.ItemsPanel>
<ListView.ItemTemplate>
    <DataTemplate>
        <DockPanel Tag="{Binding RelativeSource={RelativeSource AncestorType={x:Type Window}}}">
            <DockPanel.ContextMenu>
                <ContextMenu DataContext="{Binding Path=PlacementTarget.Tag, RelativeSource={RelativeSource Self}}">
                    <MenuItem Header="{Binding SelectedFile.FileEntity.Name}" />
                    <Separator />
                    <MenuItem Header="Open" Click="MenuItemRun_Click" />
                    <MenuItem Header="Save File As..." Click="MenuItemSave_Click"/>
                </ContextMenu>
            </DockPanel.ContextMenu>
```

2.2.3 User Interface Design

For the user interface we choose to loosely follow material design principles. There colors are monochromatic, there is one main color used for titles, important UI elements and accents. Shadows are used to project hierarchy between dialogs and various UI elements. Images and content are spread edge to edge and moving UI elements have organic motion.

The realization of this type of graphical user interface is made possible with the Material Design In XAML Toolkit [3], which is made after Google's original material design principles and keeps closely to its visual styles.

Most interaction routines are asynchronous and run independently from the UI. For example, multiple downloads can run in parallel, while the UI remains functional and notifies the user of changes happening in the background.

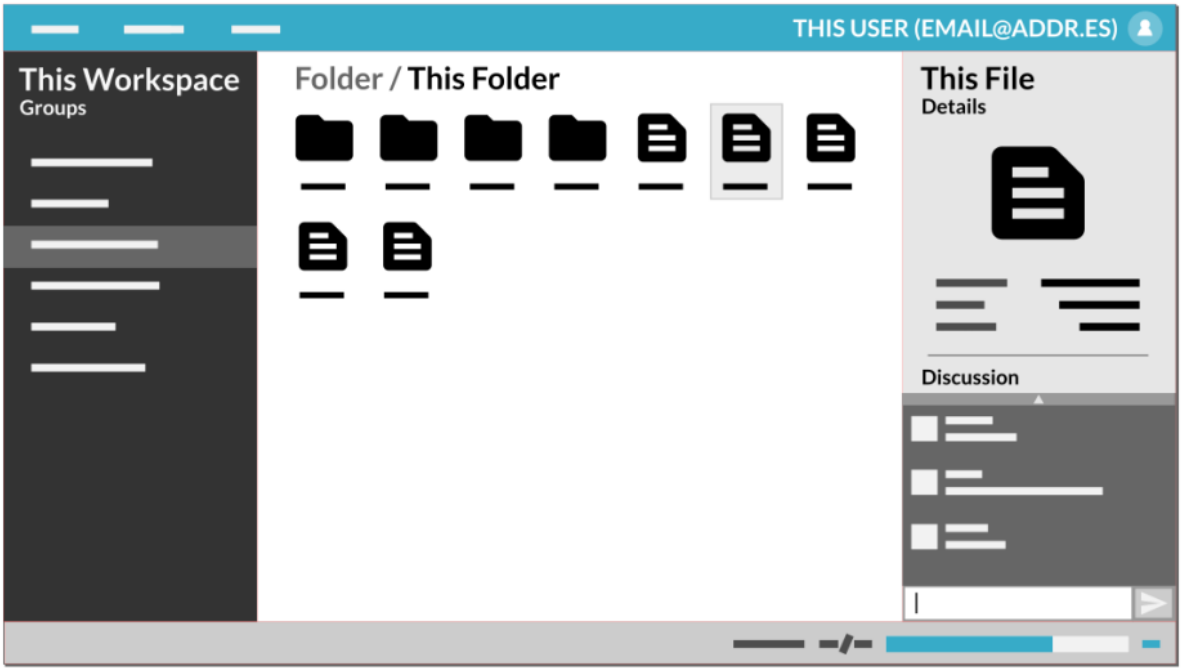


Figure 6: Mockup of the Desktop client main view

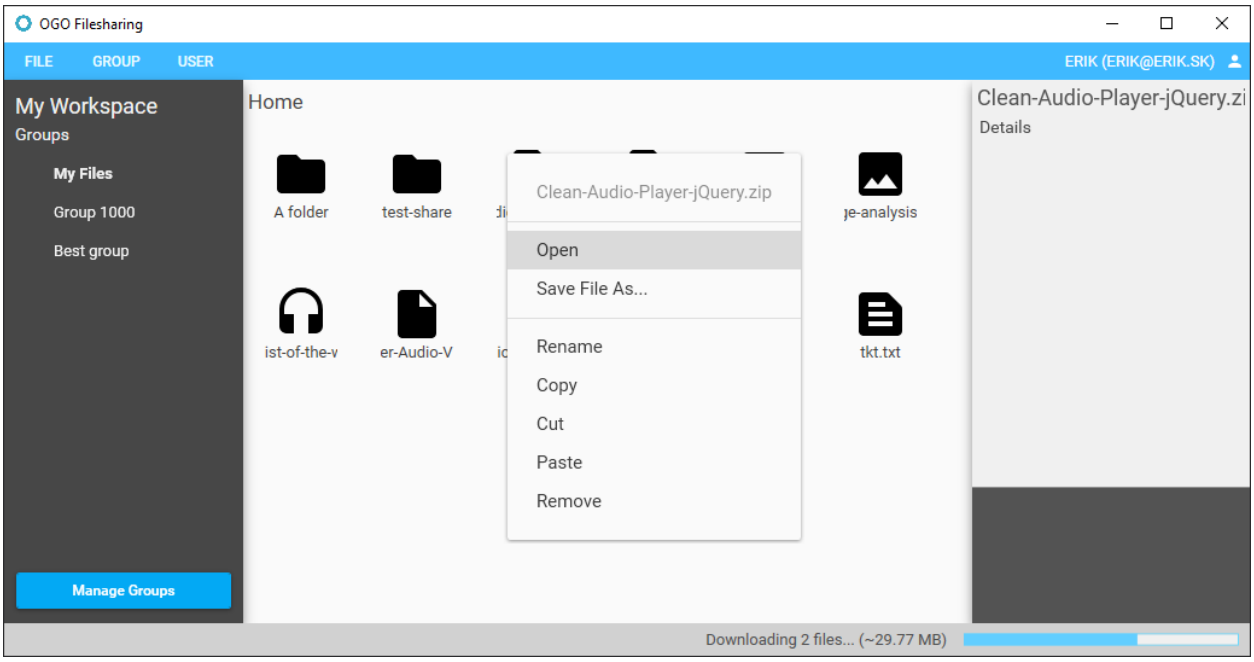


Figure 7: The main window in development

Figure 8: Login view

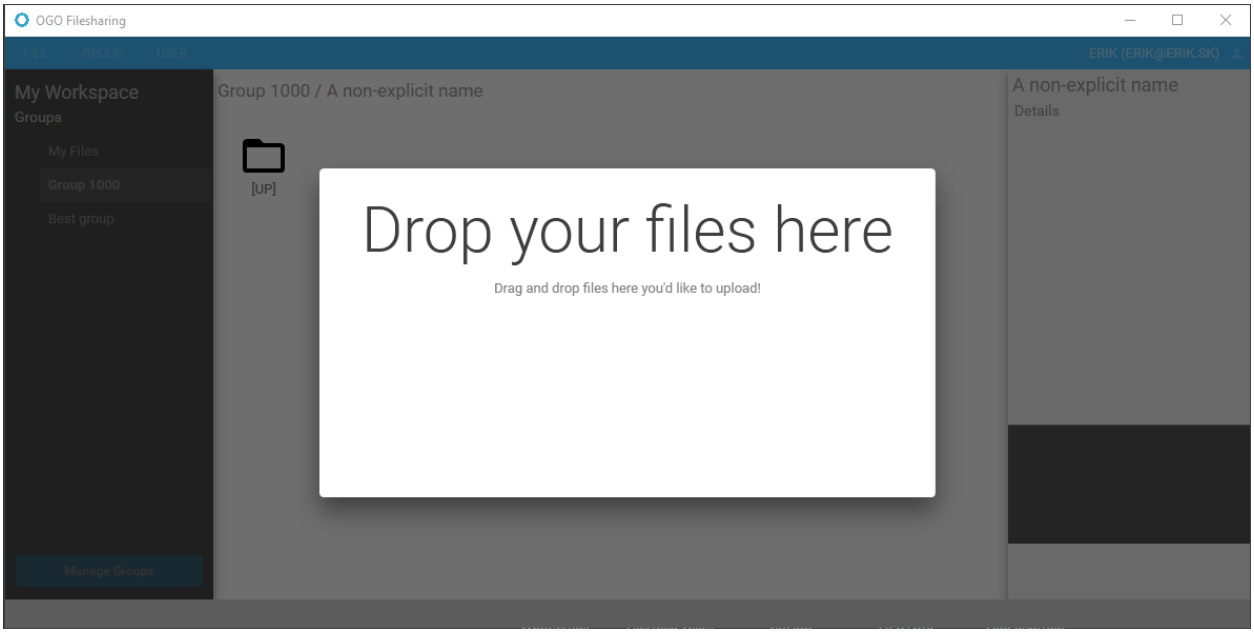


Figure 9: Drag and drop popup dialog

2.2.4 Services

The application uses an API connector class called ApiService to send requests to the server, process the responses and deserialize the response contents back to the original model classes and types.

A possible improvement would be to, instead of instantiating a new HttpClient on every request, keep reusing a single instance. This would speed up access by not having to re-initiate a TCP connection every time a request is sent and would also use up less sockets on the client OS.

2.3 Web Client

At first, we were exposed to the approach of web client being separate ASP.NET MVC project, which would access the API. After a while one of the group members came with an idea of having API and web application in one project.

By choosing the approach of having web API and web application in one project, we would eliminate the need of our web client to store bearer token and access API from MVC controllers. The approach of having web client separated from API would be less error prone.

Final choice was having web application implemented together with API in one project. One of the big factors for making this decision is the fact that we planned to use a lot of jQuery ajax requests, which would call API endpoints. This meant, that we could use Cookie authentication along with bearer token to access the API. This allowed us to seamlessly authenticate using cookies and MVC controllers and still use API endpoints.

If we were to have these separated, we would have to:

- Implement bearer token authentication JavaScript

  or

- Duplicate API requests in MVC controllers

Neither of the options was optimal, but if we were to decide, we would implement bearer token authentication in JavaScript.

2.3.1 Used client-side libraries

In our web application we used multiple client-side libraries, which made our work easier and code better. For their installation we used feature called client-side libraries in visual studio.

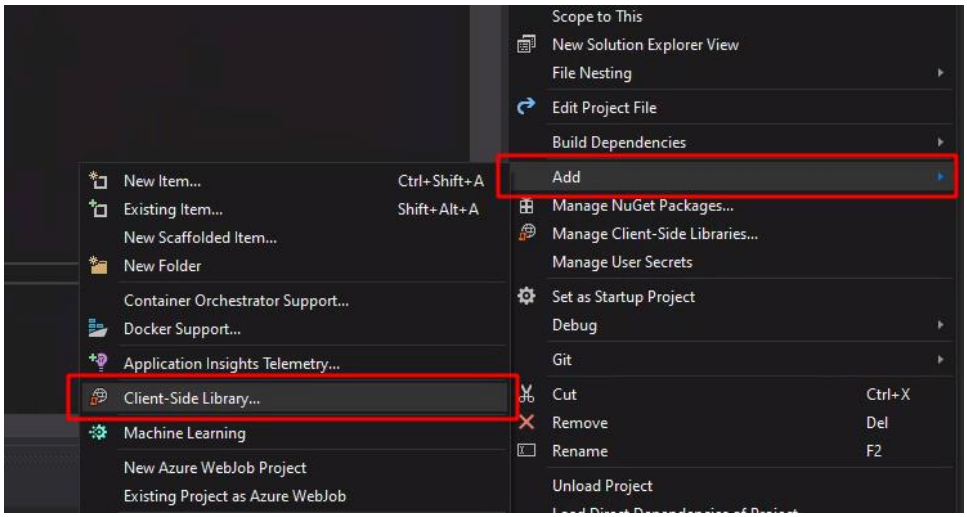


Figure 10: Adding of Client-side library in Visual Studio

These dependencies are stored in file called **libman.json**. This file is in root folder of our WebApp.

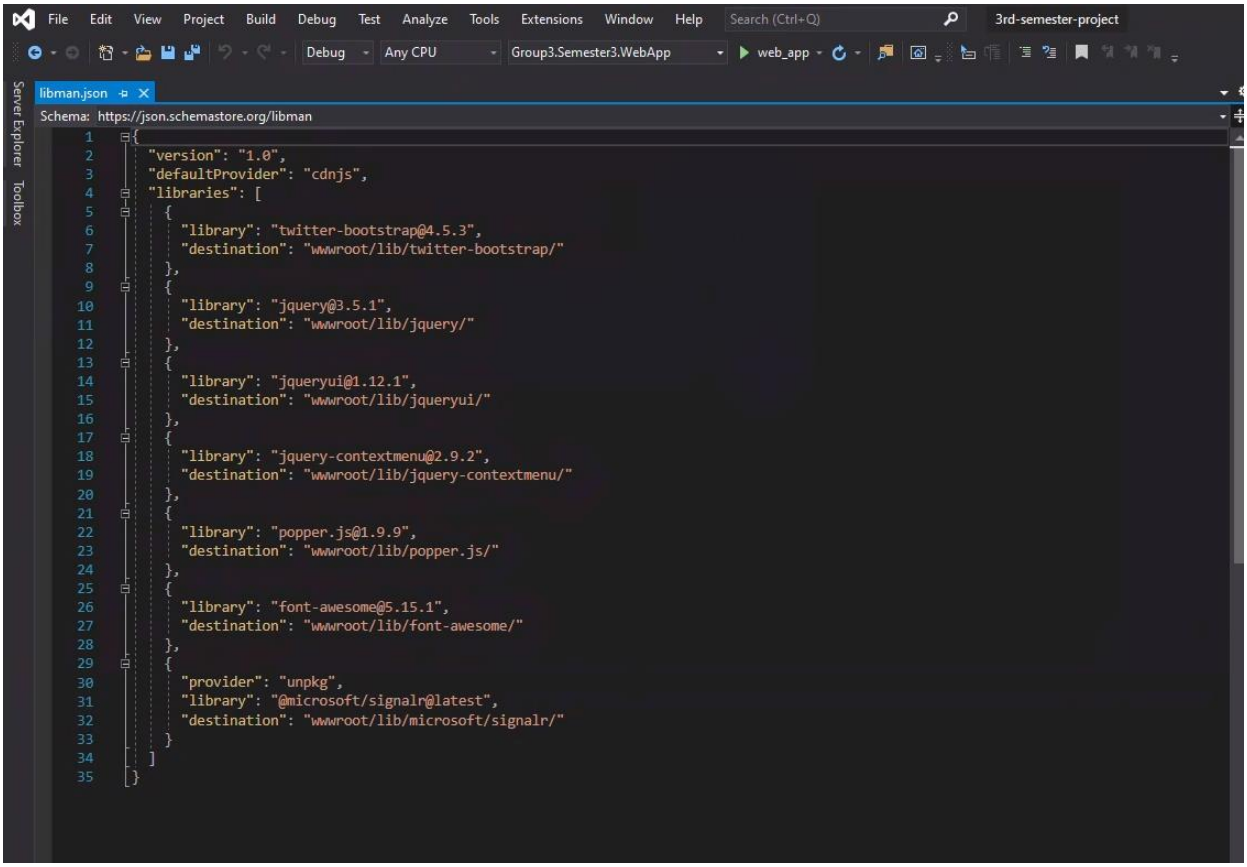


Figure 11: Contents of libman.json

Dependencies are then installed during the build of the app. This feature is very useful; all those files do not have to be stored in git repository and can be safely ignored.

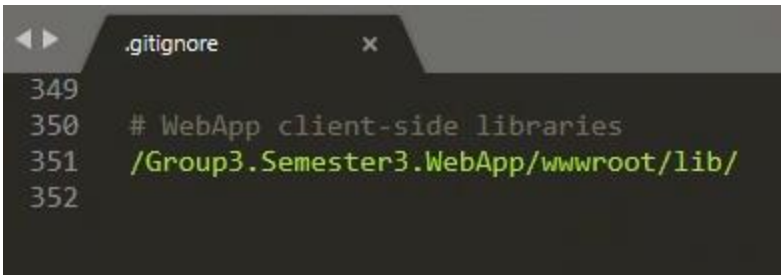


Figure 12: library root ignored from git

2.3.1.1. jQuery

jQuery is JavaScript library with many useful features which ease the work with html elements and helps with making websites more dynamic. It is also a dependency for some of the libraries we used.

More information: https://jquery.com/

2.3.1.2. Bootstrap

Bootstrap is the most popular CSS and JavaScript framework. It is useful when it comes to styling and formatting the design of the web. It contains numerous built-in styles and components.

More information: https://getbootstrap.com/

Best examples are modals we are using:

To create a modal, we need to write html code based on the examples provided in bootstrap documentation. [8]

```
<div class="modal fade" id="downloadFileModal" tabindex="-1" role="dialog" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title">Downloading <span id="download-file-name"></span></h5>
                <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <p>Your file is ready to download</p>
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-danger" id="cancel-download">Cancel</button>
                <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>
            </div>
        </div>
    </div>
</div>
```

Figure 13: HTML code snippet of download file modal

Afterwards it's just a matter of retrieving this element in JavaScript and calling modal() function.

```
function startFileDownload(file, link) {
    let $downloadLink = $('#downloadLink');
    $downloadLink.remove();

    let downloadLink = "<a href='" + link + "' download='"+ file.name +"' id='downloadLink'>Click here to download</a>"
    $('#downloadFileModal .modal-body').append(downloadLink);

    $('#downloadFileModal').modal();
}
```

Figure 14: JavaScript code snippet with download file logic

2.3.1.3. jQuery context menu

jQuery context menu is JavaScript library that uses jQuery to create beautiful and simple context menus. These menus are known as “right click menus”. This library allows for multiple options when creating these menus.

Figure 15: Context menu for file

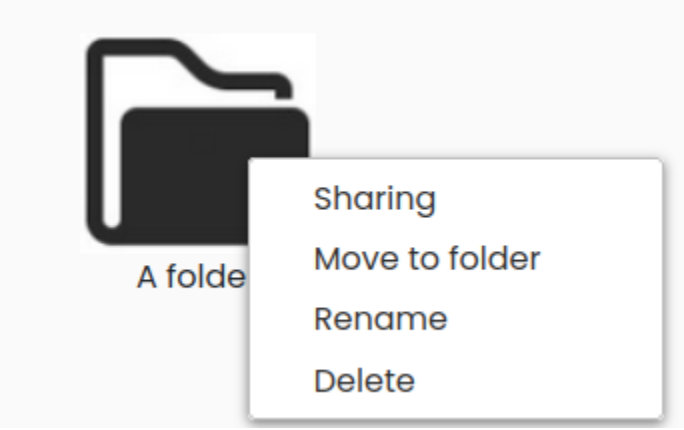


Figure 16: Context menu for folder

Context menus can be created either statically or dynamically using build function callback. We use build function callback to show different options based on file types and user permissions.

Here is the example of including view option only for files that are available for preview.

```
$.contextMenu({
    selector: '.file',
    build: function($trigger, e) {
        let items = {};

        let fileName = $trigger.find('.file-name').text();
        let classes = $trigger.attr('class');
        if (endsWithAny(previewFiles, fileName)) {
            let view = {
                view: {
                    name: "View",
                    callback: function (key, opt) {
                        let $element = opt.$trigger;
                        let id = $element.attr('id');
                        let fileName = $element.find('.file-name').text();

                        previewFile(id, fileName);
                    }
                }
            }
            Object.assign(items, view);
        }
```

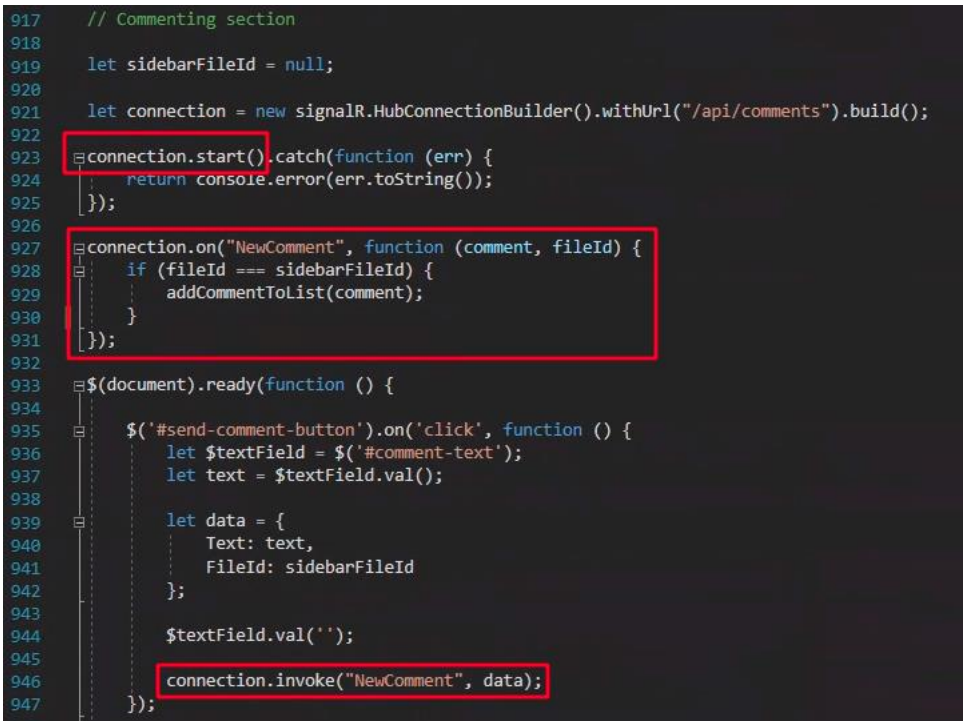
Figure 17: Code snippet with definition of Context menu in JavaScript

More information: https://swisnl.github.io/jQuery-contextMenu/

2.3.1.4. SignalR

SignalR is ASP.NET library that simplifies the process of adding real-time web functionality to applications.

Our use case for SignalR were comments. We realized we could use SignalR to update them in real time. From the JavaScript side it is just about connecting to SignalR hub using classes offered by SignalR JavaScript library and defining some callbacks.



```
// Commenting section

let sidebarFileId = null;

let connection = new signalR.HubConnectionBuilder().withUrl("/api/comments").build();

connection.start().catch(function (err) {
    return console.error(err.toString());
});

connection.on("NewComment", function (comment, fileId) {
    if (fileId === sidebarFileId) {
        addCommentToList(comment);
    }
});

$(document).ready(function () {

    $('#send-comment-button').on('click', function () {
        let $textField = $('#comment-text');
        let text = $textField.val();

        let data = {
            Text: text,
            FileId: sidebarFileId
        };

        $textField.val('');

        connection.invoke("NewComment", data);
    });
```

Figure 18: SignalR implementation in JavaScript

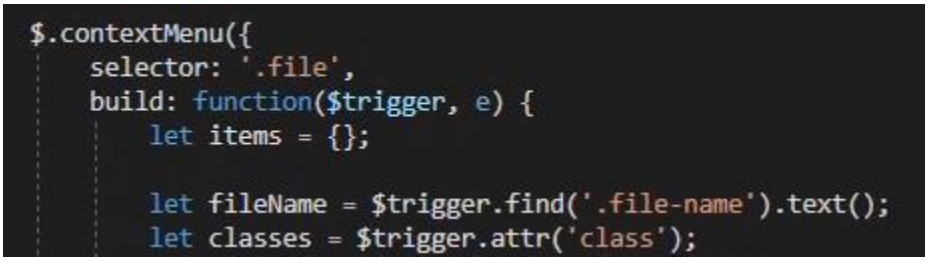
SignalR JavaScript library is easy to use. These ~ 30 lines of code handle real-time commenting on web. It is also secure, since all the authorization is done on backend.

2.3.2 JavaScript used in Browse view

Browsing (user, group, shared) files is coded as one view without reloading the page on user actions. This is possible thanks to jQuery. Everything is loaded using ajax requests and then dynamically changed.

Its lifecycle can be explained by rename file use case.

First, a context menu entry is created when user has “write” permission



```
$.contextMenu({
    selector: '.file',
    build: function($trigger, e) {
        let items = {};

        let fileName = $trigger.find('.file-name').text();
        let classes = $trigger.attr('class');
```

Figure 19: Definition of Context menu in JavaScript

```
if (currentUser.permissions.hasWrite) {
    let standardItems = {
        rename: {
            name: "Rename",
            callback: function (key, opt) {
                showRenameFileModal(key, opt);
            }
        },
        delete: {
            name: "Delete",
            callback: function (key, opt) {
                showDeleteFileModal(key, opt);
            }
        }
    }

    Object.assign(items, standardItems);
```

Figure 20: Definition of context menu items: Rename and Delete

In each context menu entry, there is callback function defined that is called when user clicks on menu entry.

It contains:

- **key -** index of menu entry object, in this case it's a string "rename"
- **opt -** object that contains valuable information about how the rename was triggered. We can extract information about file from this variable

For rename it calls function showRenameFileModal() which opens modal to rename a file.

```
function showRenameFileModal(key, opt) {

    let $element = opt.$trigger;
    let fileId = $element.attr("id");

    let fileName = $element.find('.file-name').text();

    $("#file-name").val(fileName);
    $("#file-id").val(fileId);

    $("#renameFileModal").modal();

}
```

Figure 21: Logic of retrieving file ID and Name in JavaScript

Before opening the modal, file name and id are loaded into the form so we can retrieve this information when user confirms file name change.

```html
<div class="modal fade" id="renameFileModal" tabindex="-1" role="dialog" aria-labelledby="exampleModalLabel" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="exampleModalLabel">Rename file</h5>
                <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <form>
                    <div class="form-group">
                        <label for="file-name" class="col-form-label">File name:</label>
                        <input type="text" class="form-control" id="file-name">
                    </div>
                    <div class="form-group">
                        <input type="hidden" id="file-id">
                    </div>
                </form>
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>
                <button type="button" class="btn btn-primary" onclick="renameFile()">Save</button>
            </div>
        </div>
    </div>
</div>
```

Figure 22: HTML Structure of rename file modal

Save button has onclick defined. It calls renameFile() function. This function retrieves data from the form and sends ajax request to API. On success it finds the file element and updates its name. On error it displays the alert.

```javascript
function renameFile() {
    let fileName = $("#file-name").val();
    let fileId = $("#file-id").val();

    let url = renameUrl;

    let file = {
        Id: fileId,
        Name: fileName
    }

    $.ajax({
        url: url,
        data: JSON.stringify(file),
        type: "PUT",
        contentType: "application/json",
        success: function (result) {
            let id = result.id;
            let fileName = result.name;

            $("#renameFileModal").modal("hide");

            let $fileNameElement = $('#' + id).find('.file-name');

            $fileNameElement.text(fileName);
        },
        error: function (result) {
            alert(result.responseText);
        }
    });

}
```

Figure 23: Implementation of renameFile() function in JavaScript

3. Domain Model + Relational model

Domain model shows all the business classes that we work with in the project. Since we were working with agile methodology, our initial domain model was far from its latest version.

At first, since we started with login and registration, we had only our User class and a lot of helper models around it. Then we created a FileEntity to be able to upload a file. This was sufficient for two sprints, after that we introduced user groups and file versioning, then we added an option to comment on a file and to share a file. In the process, we added plenty of field to our classes, for example, we added "IsFolder" and "IsShared" booleans to FileEntity. The latest version of our domain model can be seen below:

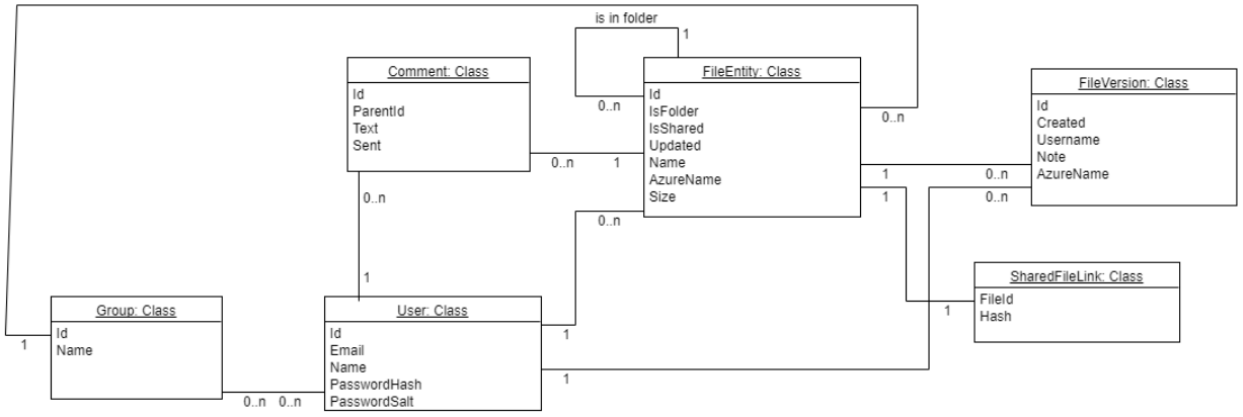


Figure 24: Final data model

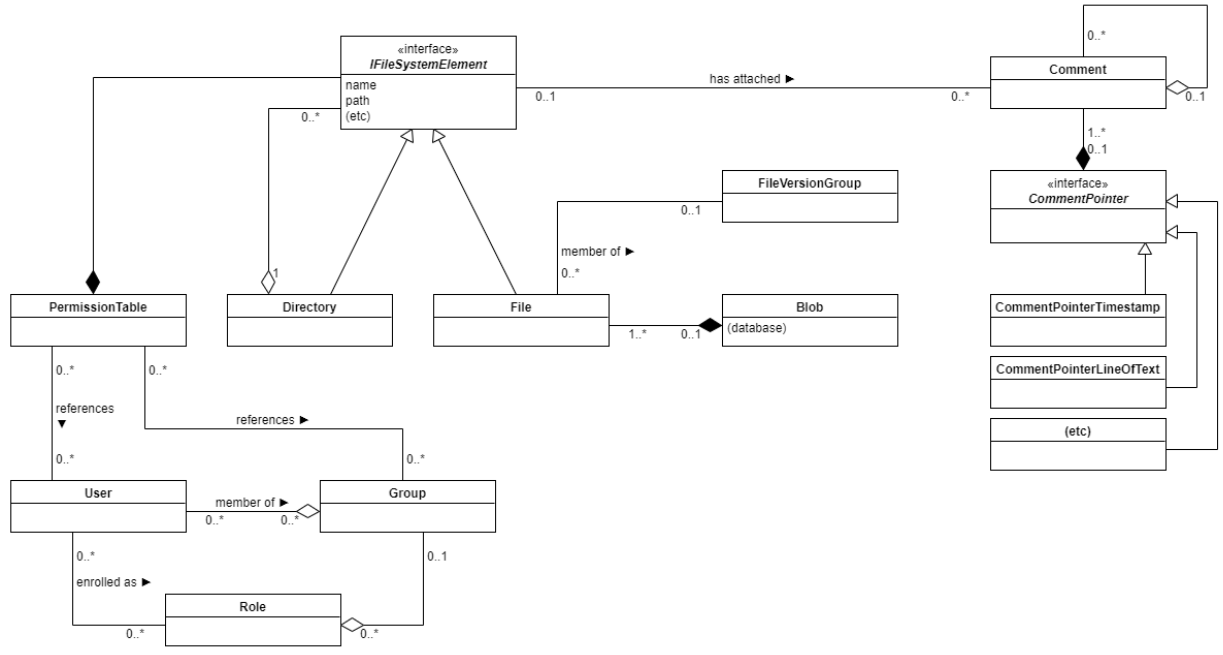


Figure 25: Initial version of the domain model

Part II. Technologies

1. Database

1.1 Possible Database Technologies

The two main options we could have chosen between were either a relational or a non-relational database (NoSQL).

If we were to choose a non-relational database, the most popular choice could have been MongoDB. If, however we were to choose a relational database, we have had a bit more options, namely MySQL (MariaDB), Microsoft SQL Server, Oracle Database or PostgreSQL.

For the ORMs we were choosing between Dapper and Entity framework.

Dapper is a micro ORM, that only helps with mapping results into classes. It's easy to use and set up but doesn't offer as much as Entity framework. Entity framework is much more complex and provides a lot of features.

1.2 Reasoning over them and our final choice

While a non-relational database could have been used for our purposes, we decided against it as none of our group members were familiar with it and we did not have enough time to learn it from the ground up. Therefore, we agreed to use a relational database.

Then, we have decided that we will either use MySQL or MSSQL, because this two were the only ones that most of our group has used before. While some of our group members had a bit more experience in MySQL, we decided to use Microsoft SQL Server, because this is the one that we have been using most for the last almost 3 whole semesters.

During the sprint 0 we were trying to setup Entity framework, but had issues getting it to work properly. Reflecting on it we realized that it was because the project was set up incorrectly.

So, we opted for Dapper, since we were comfortable with writing SQL queries ourselves.

1.3 Relational model

Our relational model was a lot different initially than it is now. Since the classes in our domain model also act as our database entities (helper model classes are not shown), the relational model mirrors the domain model. It was created with the help of SQL Management Studio and its final version is below.

Concerning database design and normalization, we kept slowly adding tables into our database as we needed them, adhering to the rules for a good database design. When looked at the relational model now, we can see that the database is at 3rd normal form right now.

The biggest challenge for us was designing tables for sharing a file between a user, which was resolved by creating a junction table between Users table and Files table, taking only the primary keys from them (UserId and FileId) as the attributes in it.

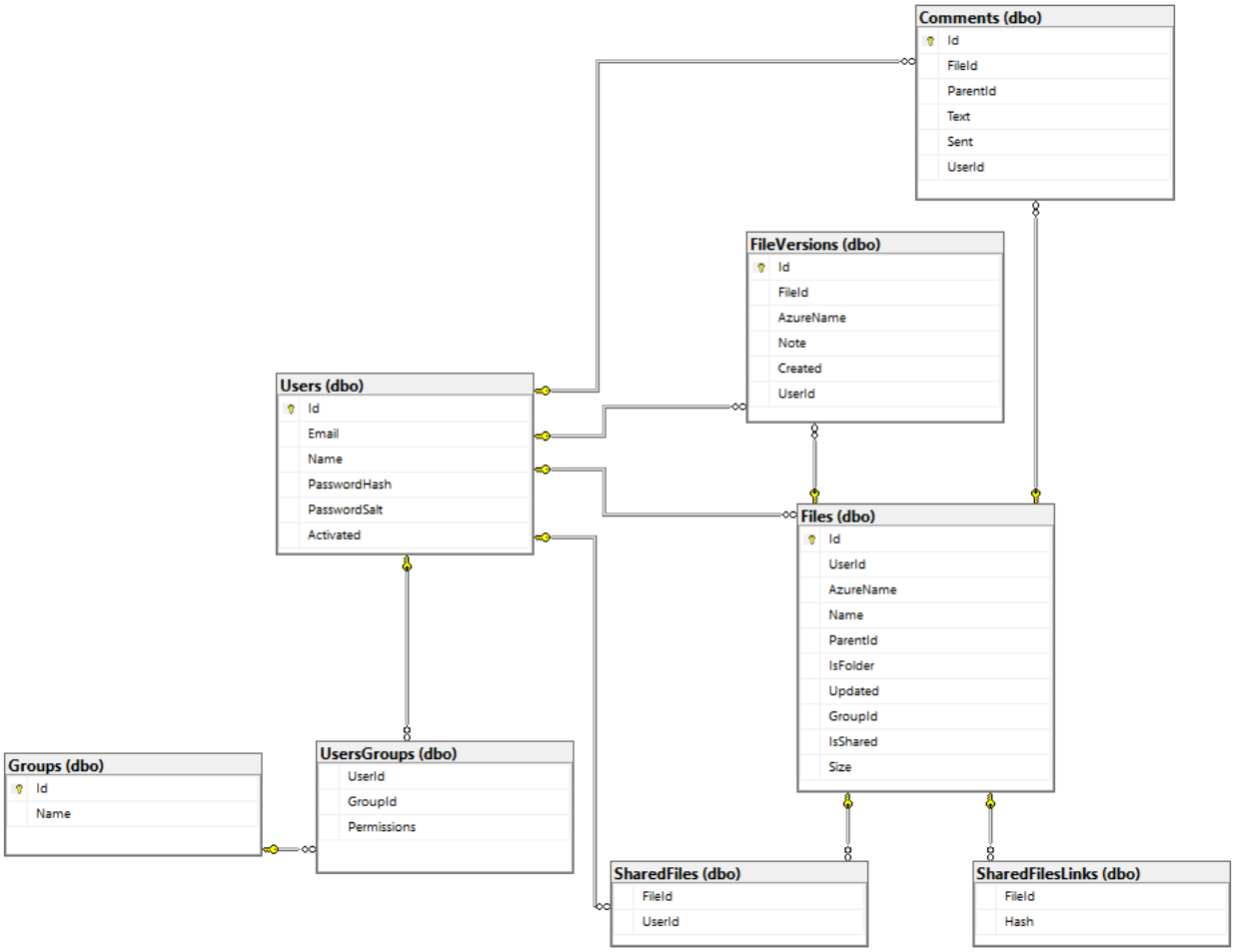


Figure 26: Final version of relation model

1.3.1 Migrations

For creating our database, we used migrations. We used FluentMigrator as NuGET package in our project, since that option that appeared the best while we did our research. With a simple "Add-FluentMigration {name}" command, our migrations were created. Fluent Migrator allows us to insert/delete data from our database and refreshes the database automatically. Each of our migration has both Up() and Down() methods, Up() serving as migration creation, Down() is for reverting the state we introduced. We did not opt for any test data insertion into the database since that was not necessary.

In our Program.cs class, there is a static UpdateDatabase(IServiceProvider serviceProvider) method that takes the Migration runner service and then runner.MigrateUp() does all the job. This method is then called in Main() method of Program.cs.

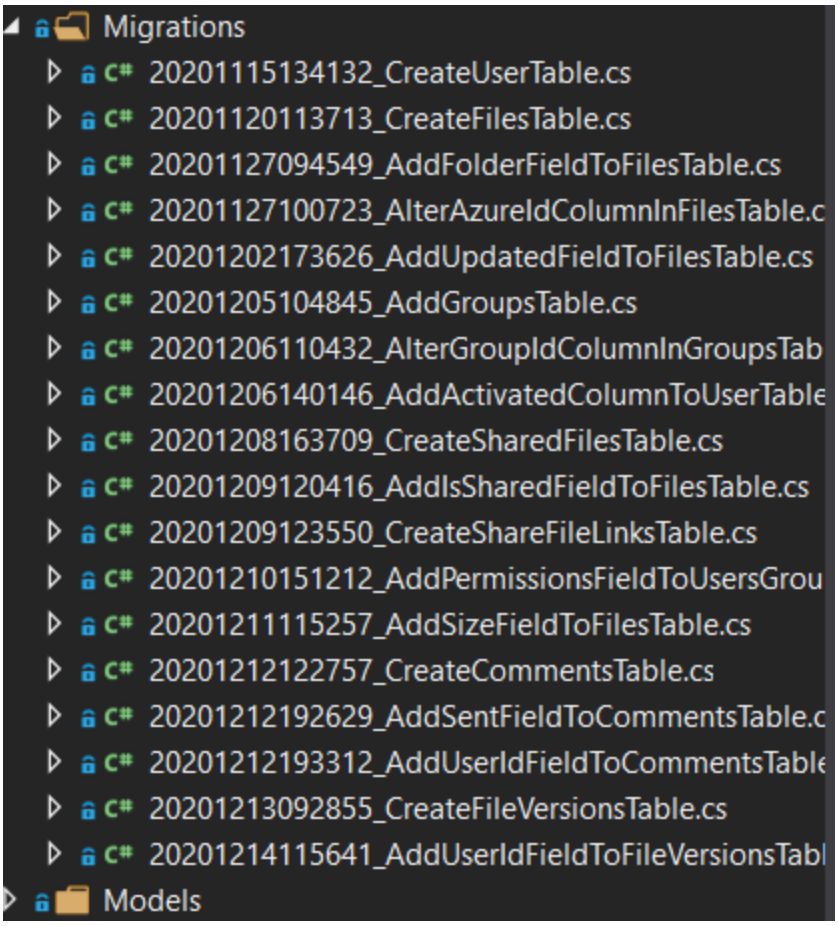


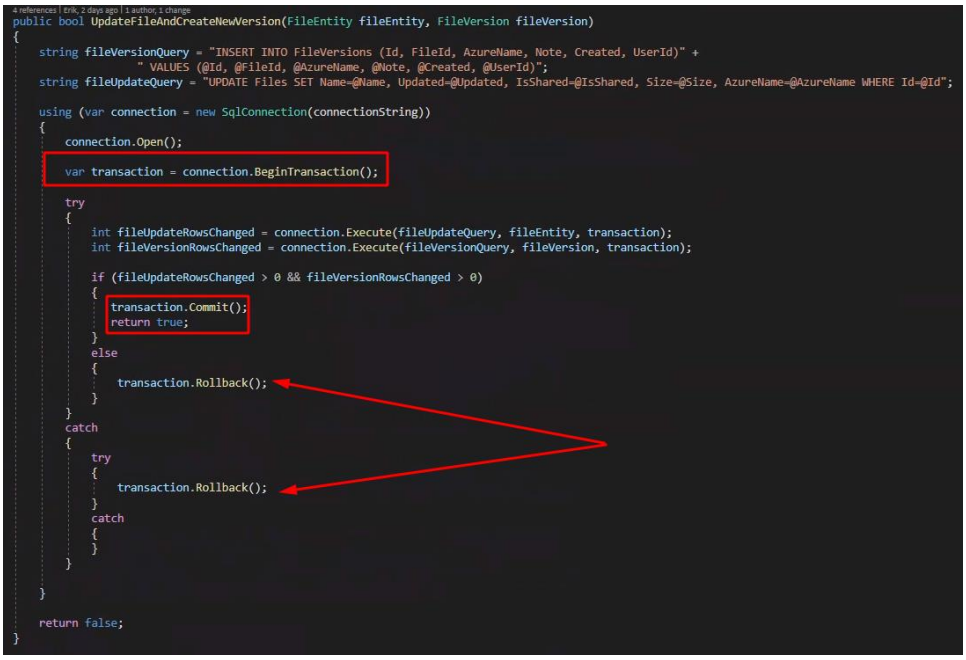
Figure 27: All created migration

```
using FluentMigrator;

namespace Group3.Semester3.DesktopClient.Migrations
{
    [Migration(20201208163709)]
    0 references | Erik, 6 days ago | 2 authors, 2 changes
    public class CreateSharedFilesTable : Migration
    {
        17 references | Erik, 6 days ago | 2 authors, 2 changes
        public override void Up()
        {
            Create.Table("SharedFiles")
                .WithColumn("FileId").AsGuid().NotNullable()
                .WithColumn("UserId").AsGuid().NotNullable();

            Create.ForeignKey()
                .FromTable("SharedFiles").ForeignColumn("UserId")
                .ToTable("Users").PrimaryColumn("Id")
                .OnDeleteOrUpdate(Rule.Cascade);
            Create.ForeignKey()
                .FromTable("SharedFiles").ForeignColumn("FileId")
                .ToTable("Files").PrimaryColumn("Id")
                .OnDeleteOrUpdate(Rule.Cascade);
        }

        17 references | mstrucka, 10 days ago | 1 author, 1 change
        public override void Down()
        {
            Delete.Table("SharedFiles");
        }
    }
}
```

Figure 28: Code snippet of migration

1.4 Transactions

We used transactions when we create a new version of a file. Since file table is also changed there are two queries that are being executed. We must make sure both are successful.



```
4 references | Erik, 2 days ago | 1 author, 1 change
public bool UpdateFileAndCreateNewVersion(FileEntity fileEntity, FileVersion fileVersion)
{
    string fileVersionQuery = "INSERT INTO FileVersions (Id, FileId, AzureName, Note, Created, UserId)" +
                              " VALUES (@Id, @FileId, @AzureName, @Note, @Created, @UserId)";
    string fileUpdateQuery = "UPDATE Files SET Name=@Name, Updated=@Updated, IsShared=@IsShared, Size=@Size, AzureName=@AzureName WHERE Id=@Id";

    using (var connection = new SqlConnection(connectionString))
    {
        connection.Open();

        var transaction = connection.BeginTransaction();

        try
        {
            int fileUpdateRowsChanged = connection.Execute(fileUpdateQuery, fileEntity, transaction);
            int fileVersionRowsChanged = connection.Execute(fileVersionQuery, fileVersion, transaction);

            if (fileUpdateRowsChanged > 0 && fileVersionRowsChanged > 0)
            {
                transaction.Commit();
                return true;
            }
            else
            {
                transaction.Rollback();
            }
        }
        catch
        {
            try
            {
                transaction.Rollback();
            }
            catch
            {
            }
        }

    }

    return false;
}
```

Figure 29: Implementation of transaction

Transaction in only committed when both queries are successful. If not, transaction is rolled back. It is also rolled back when any exception is thrown.

2. Server (services)

2.1 Rest vs SOAP

When making a web application the use of a web API service is crucial; it allows the communication between the producer and consumer side of the application. The two main possibilities we use today are SOAP and REST. SOAP (Simple Object Access Protocol) is designed by Microsoft and it has been around for many years. It is a standardized web service but it is not the most flexible. On the other hand, we have REST (Representational State Transfer), which is a newer web service that offers more flexibility and is easier to use.

2.2 Reasoning over them

SOAP is an independent service regarding language and transportation. Beside HTTP it also uses other protocols. SOAP does not return human readable results, which means more security, but makes working with it more difficult. It can work with JavaScript; however it is not easy to implement. Compared to REST, it does not have a great performance.

REST is a way more flexible service, easy to learn and use. It uses XML or JSON formats to transfer data, so the results are easily readable as well. It calls the requests using URL paths and it is easy to call from JavaScript. The transfer is only through HTTP(S) protocol. The performance of REST is way better than SOAP's. REST is more commonly used nowadays, most of today's web APIs are REST APIs.

REST will be our choice for this project, since the group was more comfortable using it and it suited our needs more.

2.3 Implementation Details

The implementation of it is shown by the method to browse files. In the FileApiController, the route is specified to be API/file/browse, it takes groupId and parentId from query as parameters, calls userService.GetFromHttpContext() to get the logged in user and then fileService.BrowseFiles() method to retrieve all the files, returning tuple types.

```
/// <summary>
/// GET: api/file/browse
/// </summary>
/// <returns>FileEntities that are owned by the user</returns>
[HttpGet]
[Route("browse")]
0 references | Erik, 1 day ago | 1 author, 3 changes
public IActionResult BrowseFiles([FromQuery] string groupId, [FromQuery] string parentId)
{
    try
    {
        var user = _userService.GetFromHttpContext(HttpContext);
        var (userModel, files) = _fileService.BrowseFiles(user, groupId, parentId);

        return Ok(new {
            user = userModel,
            files = files
        });
    }
    catch (ValidationException exception)
    {
        return BadRequest(exception.Message);
    }
    catch
    {
        return BadRequest(Messages.SystemError);
    }
```

Figure 30: BrowseFiles AP endpoint controller implementation

In the Postman app we used the request can be called easily and the returned body is readable for everyone

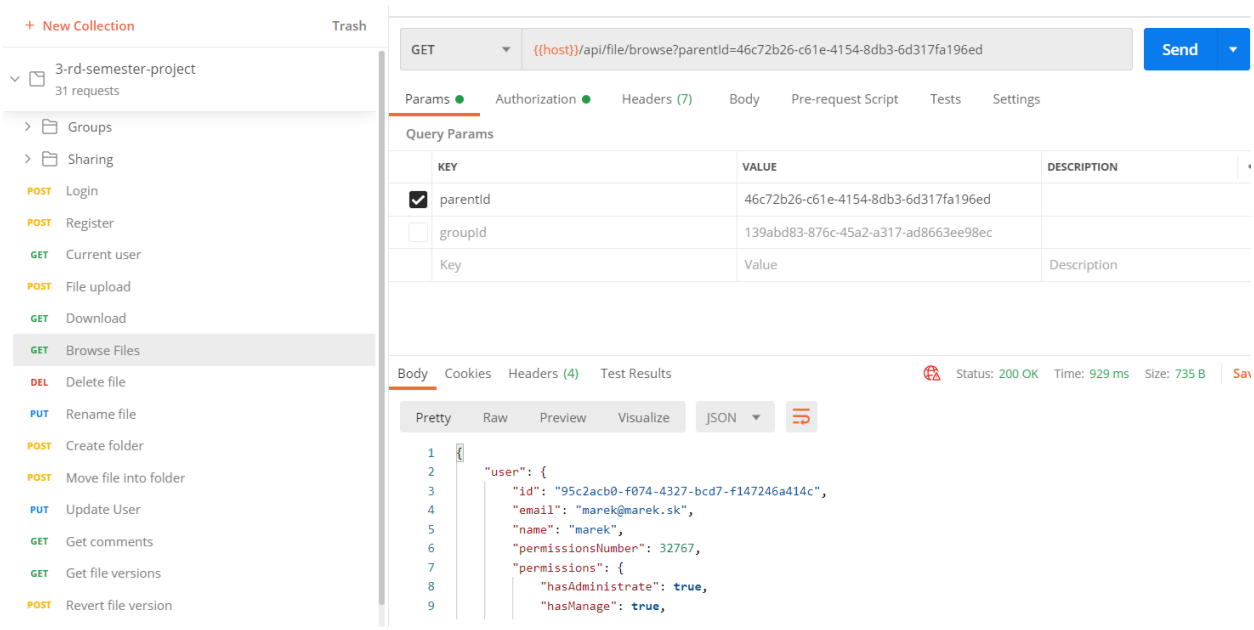


Figure 31: Screenshot of postman request with part of a result


```
"files": [
    {
        "id": "0daf63dc-bbb6-4ebe-b216-65fd48eed0ec",
        "userId": "6ca63f0c-406d-4f0d-831d-daed56ab5dd9",
        "groupId": "139abd83-876c-45a2-a317-ad8663ee98ec",
        "azureName": "b45576b1-4c1e-4c67-ba82-95dd3191fcb0",
        "name": "config.ini",
        "parentId": "46c72b26-c61e-4154-8db3-6d317fa196ed",
        "isFolder": false,
        "isShared": false,
```


Figure 32: JSON result of a request

3. Azure Services

3.1 Azure Blob Storage

The most important Azure Service we used during our project is the Azure Blob Storage. This is a BLOB based file storage offered by Azure. We choose this storage type instead of a local disk or a network storage, because of its massive scalability and high performance with a relatively low cost. We don't have to bother with the amount of storage we have as this automatically scales and even petabytes of data can be uploaded to it.

Each Azure user can have multiple “containers” and each container can have multiple “blobs” in them. Containers can be used like “folders”, but they cannot be nested inside one another and Blobs are the files which are uploaded to the service.

All interactions with this service are done through the official Azure .NET library in the WebApp. For the file uploads we are using the library’s BlobClient.UploadAsync() method which uploads the files to the Azure storage. To download the files, we are not using the build in DownloadAsync() method, as it can only download the files locally, which can be used in the DesktopClient, but is not optimal in the Web App. Therefore, we use another feature of this Blob Storage, called Shared Access Signature (SAS) tokens, which is a signed URI that points to a specific blob in a container. With this SAS tokens, we can control the expiration of download links and authorization to files as only the user who has access to a file (which is checked by the API, see paragraph 3.6.2) will be able to generate a token to their own files. By default, all files uploaded to the Blob Storage are private, therefore only the owner of the Azure account can view them or share them with other Azure users. With this SAS token, people without an Azure account can view and access files using a special signed URL.

3.2 Azure Microsoft SQL Server Database

As we chose Microsoft SQL Server for our project (see explanation in paragraph 3.1), we needed to find a service which offers a stable, reliable, and cost-effective way for hosting our database. We decided to go with the Azure MS SQL Server. It is a managed SQL Database; we don’t have to configure and manage the server; we were able to provision and start this server in under 5 minutes.

It offers the same features and capabilities as a regular self-hosted Microsoft SQL Server, with the additional thing mentioned above.

3.3 Azure App Service

The final Azure product used for our project is the Azure App Service. It is a serverless (with no direct access to a physical or virtual server), fully managed PaaS (platform as a service). It allows us to simply deploy our Web App directly from Visual Studio or via FTP. It can work with several programming languages and can run our code on Windows or Linux.

We have got a custom domain name, which we attached to this App Service instance using Cloudflare (used to manage and set up the DNS records of our domain name). It can be accessed at the following domain name: https://ogo-file.space.

Part III. Security

1. Authentication

In such application, an authentication scheme to ensure that users can only access things that are meant to is essential. We decided to use bearer authentication for the desktop client and cookie authentication for the web app. The reason is that both are widespread authentication schemes that - despite being simple - work quite efficiently. Bearer authentication works with security

tokens called bearer tokens; encrypted strings generated by the server that are sent in the header of each request which accesses protected resources. Similarly, cookies store the information for the authentication, and they are sent along with each request made in the browser.

Bearer token is generated when the user logs into the application using the desktop client. After submitting the credentials, the UI sends a request for the login through the ApiController. The login request checks if the credentials are correct, then creates a token using the JwtSecurityTokenHandler. We can specify the expiration time of the token, we set this to 7 days. With creating a new Claim, we assign the current user to the token, so each token has its user. After the token is created, the system writes it into a string and returns it to the ApiController where it gets stored as a protected string.

The cookie authentication works in a similar way in the web application, but instead of creating a Bearer token, the system generates a cookie. Most of the logic for the authentication process is predefined and can be found in the Startup.cs file. When a bearer token is created, we check if the user exists.

When the user logs out, the user gets removed from the Http context.

2. Authorization

After the user has the authentication token, it is sent together with the requests when it accesses parts of the application that require authorization. In the beginning of every request, we defined what type of authorization is needed for the request. In most parts it is either bearer token or cookie authorization, but for some requests (registration) no authorization is required.

3. Password encryption

To ensure the safety of our system, we had to find a way to store the passwords safely. Here, the common practice is using password hashing along with salting. Salting is the practice of mitigating password attacks by adding a random value, that is strong in a cryptographical sense, to the password before hashing it. This way the password hashes stored in our database are unique. This method also makes rainbow table attacks way harder.

In our system this happens in the UserService class. Here we have a private helper method, for creating the hash using the password and the salt. To create the encrypted password, we are using the System.Security.Cryptography.HMACSHA512 class. As a salt, we are using the Key attribute of the hashing, since it is a random value. In the User class we store the hashed password and the salt.

For verifying a password hash, we also have a method, which takes in both the original and the hashed password along with the salt and after hashing the original password, compares it to the one that needs to be verified.

4. SSL (HTTPS)

Another essential part of security when talking about web applications is that the communication between the server and the browser is secure. The way to ensure this is using HTTPS, which is the

secure extension of the HTTP application layer protocol. HTTPS is different to HTTP, it encrypts the data that is sent between the browser and the server, using SSL (Secret Socket Layer).

As we are using Azure App Service and Cloudflare, we did not have to spend a lot of time configuring the SSL for our Web App. We have generated an SSL Certificate that we have added to the App Service and all other configuration has been done by Azure.

5. SQL Injection

SQL injection is code injection technique that might destroy your database. It is one of the most common web hacking techniques, a placement of malicious code in SQL statements via user input. [9]

The most common SQL injection attacks are performed based on a fact, that "1=1" is always true. So, the hackers will write into input asking for a name something like "John or 1=1" which results in query as follows:

```
SELECT * FROM Users WHERE Username= John OR 1=1;
```

Figure 33: Example of SQL injected query

Since this is always true, it returns the data about the user to the hacker.

Another example is input like "John; DROP TABLE Users" which would harm our database as well.

To prevent this, we use SQL parameters. Those are values that are added to SQL query at execution time and in a controlled manner. Example of our code to insert into a database:

```
2 references | Krisztián Henrik Papp, 7 days ago | 5 authors, 8 changes
public bool Insert(User user)
{
    string query = "INSERT INTO Users (Id, Email, Name, PasswordHash, PasswordSalt, Activated)" +
        " VALUES (@Id, @Email, @Name, @PasswordHash, @PasswordSalt, @Activated)";

    using (var connection = new SqlConnection(connectionString))
    {
        user.Id = Guid.NewGuid();

        var parameters = new {
            Id = user.Id,
            Email = user.Email,
            Name = user.Name,
            PasswordHash = Convert.ToBase64String(user.PasswordHash),
            PasswordSalt = Convert.ToBase64String(user.PasswordSalt),
            Activated = true
        };

        try
        {
            connection.Open();
            int rowsChanged = connection.Execute(query, parameters);

            if (rowsChanged > 0)
            {
                return true;
            }
        }
        catch (Exception e)
        {
        }
    }

    return false;
}
```

Figure 34: Implementation of user repository insert method

6. Exception Handling

In such project, there are a lot of runtime exceptions that can occur, and it is our responsibility to handle them properly. There are various solutions on the internet on how to handle them, some of them are more correct than the others, but it also depends on the type of project and its architecture.

In our project, we introduced two helper classes that deal with exceptions, a ValidationException, since there is a huge number of validations going on in our application, and ConcurrencyException, where we display a custom message after a concurrency issue arose.

We throw all the custom exceptions in service layer, and then in controller layer, we are catching all the exceptions that arose, displaying messages we pass in the service layer for ValidationException where there are, and displaying default system error message from Messages class.

We do not want to send the given exception to the user, we strive to inform the user that something is wrong in the system, therefore we chose such approach.

```
[Route("create-folder")]
[HttpPost]
0 references | Erik, 2 days ago | 2 authors, 4 changes
public ActionResult CreateFolder(CreateFolderModel model)
{
    try
    {
        var user = _userService.GetFromHttpContext(HttpContext);
        var result = _fileService.CreateFolder(user, model);

        return Ok(result);
    }
    catch (ValidationException exception)
    {
        return BadRequest(exception.Message);
    }
    catch
    {
        return BadRequest(Messages.SystemError);
    }
}
```

Figure 35: Implementation of create-folder file API endpoint

Part IV. Patterns

1.1 Repository pattern and dependency injection

Dependency Injection (DI) is a technique in which an object receives other objects it depends on. In DI there are two subjects: Clients and Services. Services are being injected into clients. Clients thus depend on services.

In our web application all classes in repository, business logic and controller layer are subjects in DI. Either as clients or services. The usual approach is that repositories are being injected into business layer services and those services are injected into controllers.

We used DI to ensure low coupling and to eliminate the need of creating new instances of services somewhere in the code.

To ensure there are no cross dependencies we made sure that services and clients could only depend on services from lower layer. The only exception is AccessService, which some of the services in its layer depend on.

1.2 Concurrency

Concurrency handling was one of the requirements since it is becoming a norm nowadays, because of the ability of our computers to perform multiple computations concurrently. Concurrency must be properly handled with either optimistic or pessimistic approach to make sure that the data we input to the system are also shown properly in the output.

1.3 Optimistic vs pessimistic concurrency

There is a major difference in how concurrency is handled, and it differs in a way how the critical section of the code is maintained while concurrent transactions happen there.

Optimistic concurrency works based on the idea that everything stays unlocked until there is the trigger to change something (for example in the database table). When this trigger arises, the row is reread, and it is checked whether it has changed since the last read. If that is the case, the update fails and the whole transaction must happen again.

Pessimistic concurrency locks the part of the code as the user approaches it, and just after the operations are performed and saved, it is unlocked. If this happens simultaneously in two different instances, after the second user saves his changes, he gets notified of new changes and is asked to either revert or overwrite them (if possible).

1.4 Our concurrency – example

There are numerous concurrency issues that come from the scope of our project. The example we want to talk about here deals with editing of text files by two users at the same time.

After researching concurrency and looking at our found one, we decided to go for optimistic concurrency, since it did not make sense to us to lock the whole editing window for second user once user one has it opened. Therefore, we went for optimistic concurrency, which allows both users to edit the same file simultaneously and checks for the concurrency are made only after the save/delete buttons are pressed. It checks for the timestamp when the file was updated in the database and if it detects an update in the process, there comes an alert message asking the second user to either overwrite or revert the changes to the state user one left the file at.

```
public FileEntity UpdateFileContents(UpdateFileModel model, UserModel user)
{
    var file = _fileRepository.GetById(model.Id);

    if (file == null)
    {
        throw new ConcurrencyException("File was deleted by another user");
    }

    _accessService.hasAccessToFile(user, file, Permissions.Write);

    if (!model.Overwrite)
    {
        var result = DateTime.Compare(model.Timestamp, file.Updated);

        if (result <= 0)
        {
            throw new ConcurrencyException("File was changed by another user. Please try again");
        }
    }

    byte[] byteArray = Encoding.ASCII.GetBytes(model.Contents);
    var contentStream = new MemoryStream(byteArray);

    var containerClient =
        new BlobContainerClient(
            _configuration.GetConnectionString("AzureConnectionString"),
            _configuration.GetSection("AppSettings").Get<AppSettings>().AzureDefaultContainer);

    containerClient.CreateIfNotExists();

    var newVersion = new FileVersion()...;

    file.AzureName = Guid.NewGuid().ToString();
    file.Updated = DateTime.Now;
    file.Size = contentStream.Length;

    var success = _fileRepository.UpdateFileAndCreateNewVersion(file, newVersion);

    if (success)...
    else
    {
        throw new ValidationException("Failed to update a file");
    }

    return file;
}
```

Figure 36: Code snippet containing implementation of row versioning to solve concurrency issue

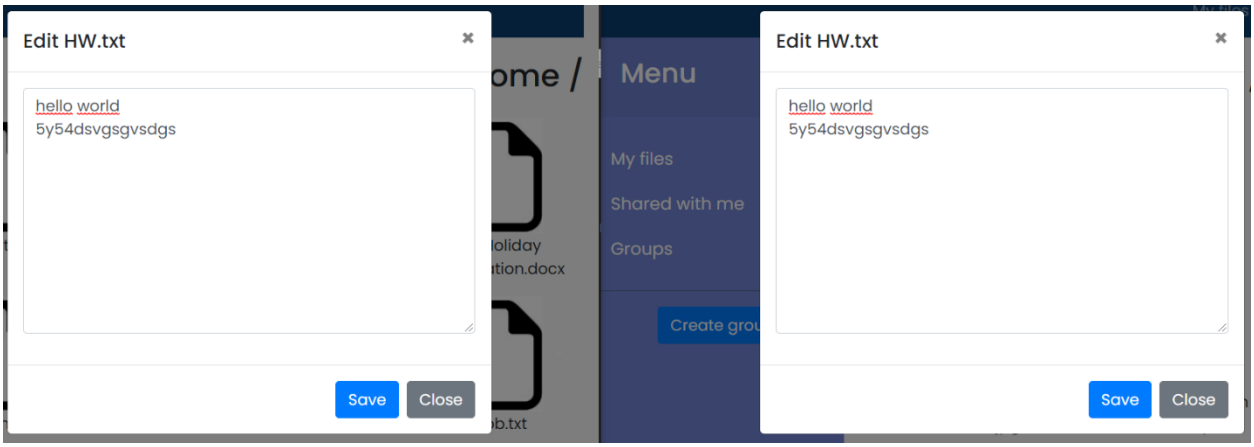


Figure 37: Editing the same file from different accounts

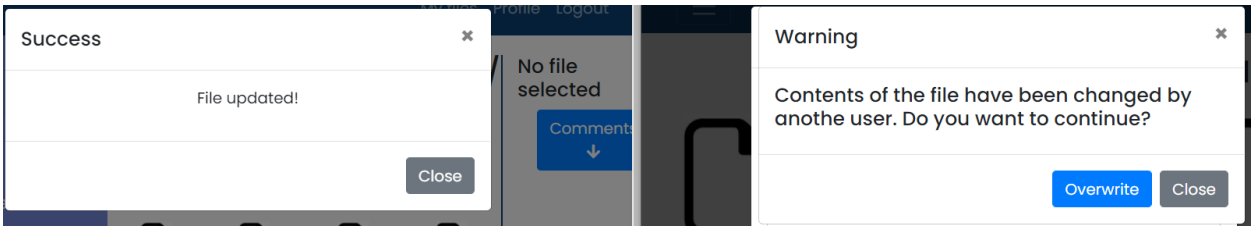


Figure 38: A warning comes up if another user already saved the file

Part V. Other Interesting Details

1. Permissions

When it comes to permissions, we chose approach similar to Unix file system permissions. Our implementation uses number stored in the database representing file permissions. Each bit represents one permission.

As of writing this, there are only four bits used. We can scale permissions as we like in the future.

We can use these values as permissions:

```
1000 0000 (128) - Administrate
0100 0000 (64)
0010 0000 (32) - Manage
0001 0000 (16)
0000 1000 (8)
0000 0100 (4)
0000 0010 (2) - Write
0000 0001 (1) - Read
```

Figure 39: Binary representation of permissions

```
37 references | Erik, 9 days ago | 1 author, 1 change
public static class Permissions
{
    // Permissions
    public const short Administrate = 128;
    public const short Manage = 32;
    public const short Write = 2;
    public const short Read = 1;
}
```

Figure 40: Implementation of class containing permissions

If user has all these permissions, it is stored in the database as sum of these numbers. (In this case 163).

We created helper class to convert this number into Booleans, so it is easier to work with in the front-end.

```
namespace Group3.Semester3.WebApp.Helpers
{
    2 references | Erik, Less than 5 minutes ago | 1 author, 2 changes
    public class PermissionHelper
    {
        0 references | Erik, Less than 5 minutes ago | 1 author, 2 changes
        public bool HasAdministrate => (PermissionsNumber & Permissions.Administrate) != 0;
        0 references | Erik, Less than 5 minutes ago | 1 author, 2 changes
        public bool HasManage => (PermissionsNumber & Permissions.Manage) != 0;
        0 references | Erik, Less than 5 minutes ago | 1 author, 2 changes
        public bool HasWrite => (PermissionsNumber & Permissions.Write) != 0;
        0 references | Erik, Less than 5 minutes ago | 1 author, 2 changes
        public bool HasRead => (PermissionsNumber & Permissions.Read) != 0;

        5 references | Erik, Less than 5 minutes ago | 1 author, 2 changes
        public short PermissionsNumber { get; set; }
    }
}
```

Figure 41: Implementation of permission helper

To define whether permission number contains certain permission, we are using bitwise AND operator. This operator returns 0 if permission is not present in permission number.

We are also using this logic in AccessService currently permissions are implemented for group members.

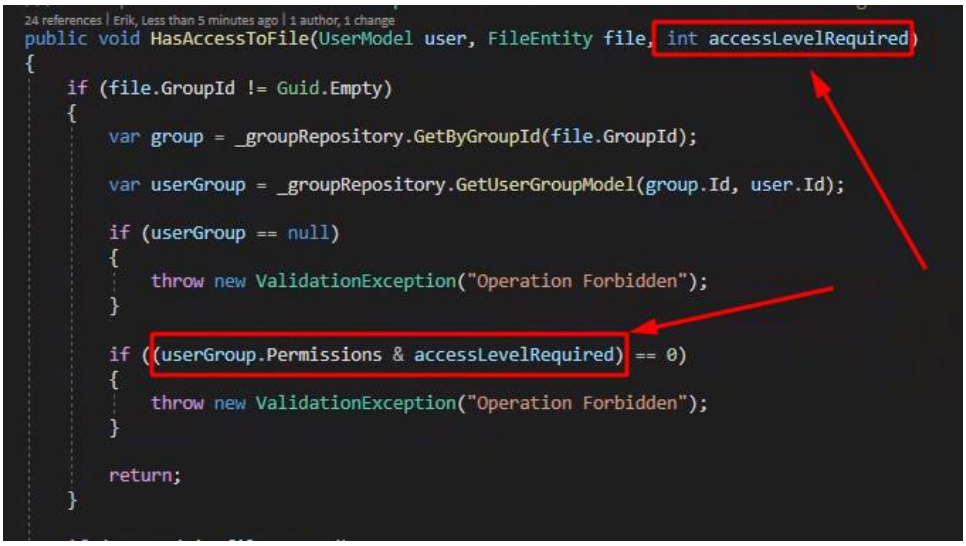


Figure 42: Implementation of permission check

This approach to access service gives us great flexibility when checking permissions for different operations.

Here are few examples.

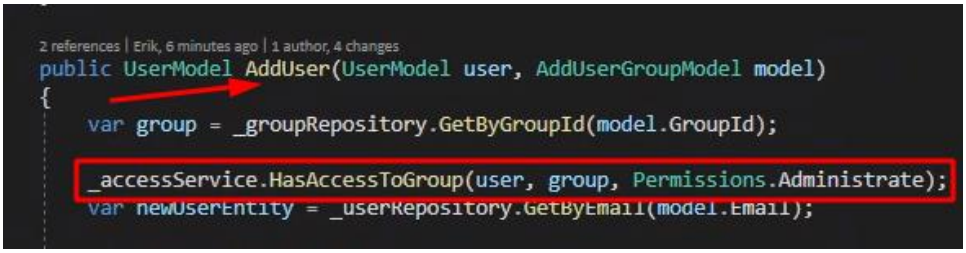


Figure 43: Usage of Administrate permission when adding user to a group

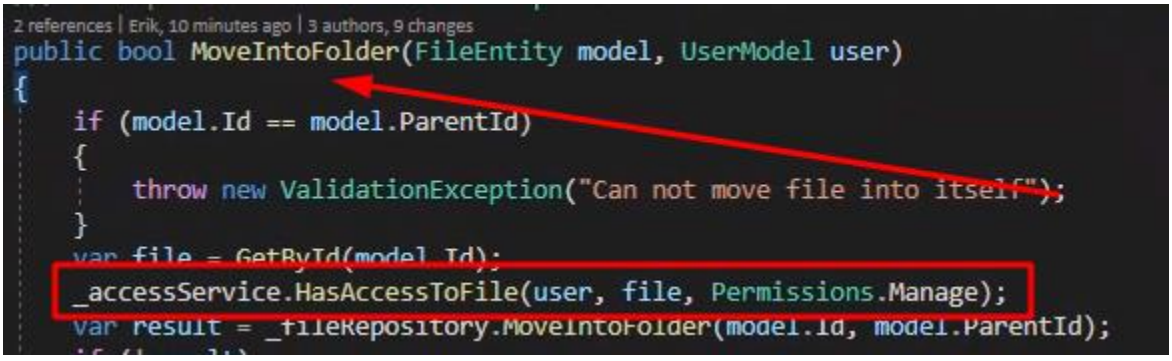


Figure 44: Usage of Manage permission when moving a file

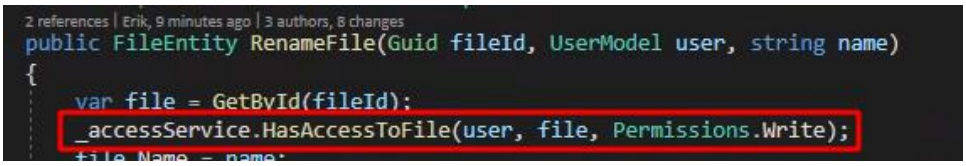


Figure 45: Usage of Write permission when renaming of file

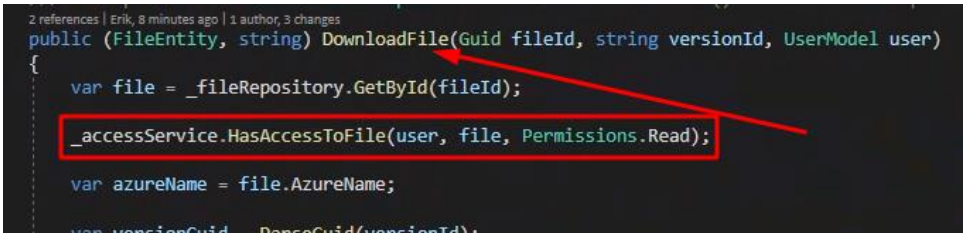


Figure 46: Usage of Read permission when downloading a file

2. Version Control System

In the beginning, we had to make decision on Version Control System. We have settled on Git, as it offers more advanced branching and repository management features than SVN.

As this has been settled, we had to choose a Git server provider. We choose GitHub as we were more familiar with it and even though Azure DevOps has more features, we agreed that the functionalities GitHub offers are enough for us.

We agreed to follow the GitHub Flow workflow, which is an improved but lightweight variant of the original Git Flow. GitHub Flow is based upon branching, where the main rule is that anything on the main branch is stable and can be deployed. Therefore, we have agreed to use "feature branches", which are all based upon the main branch and contain implementation of a single User Story (in case of bigger user stories, they might got split into several branches, but the main rule was: 1 Branch = 1 User Story). In our repository, we had three kinds of branches:

- There is one "main" branch which only contains stable code
- There are "feature/new-feature" branches, which contain the implementation of user stories
- Finally, the "fix/something-fixed" branches, which were used to have minor fixes.

Every branch only exists until it is merged, then it is automatically deleted. We have used GitHub Pull Requests to merge any branch into main and regular Git merges to move some code from any branch to another.

There were also Branch Protection rules set up for our main branch, which were protecting it against accidental pushes or merges. It was forbidden to push local changes directly to main, they always had to be committed and pushed to a feature or fix branch and then merged into main using a Pull Request. We also set up the Pull Requests in a way, that every one of them had to be reviewed by at least 2 people in the group in order to make sure everyone knows about every change made in the repository.

3. Testing

3.1 Unit tests

The team did not fulfill the requirement for test first development. Initially we had troubles designing our tests since we did not even know how we were going to implement the registration and login functionality and then there was a lot of dependency injection in our project and we could not figure out how to write those tests properly, since we have not even been taught about these issues. After reading up on the internet, we found a possible solution in mocking.

3.2 Mocking

We tried also mocking for our tests, using Moq library as our tool. Classes can be mocked together with their outputs which would verify the test, but we unfortunately were not able to do even that, leaving our test-first approach behind.

Since then, everything was manually tested. The group has agreed to try and find a way to implement tests until the exam date, so that it will be able to present those as well.

3.3 Manual testing

We have done a lot of manual testing for reasons mentioned above. We started our application and tried a lot of negative tests, inputting invalid values into the fields, tried to upload files with various extensions, tried to get into user groups we had no access to etc.

3.4 Postman - testing and documenting API

During implementation, it has occurred many times, that we finished the backend, but we haven't had a front end ready and we needed a way to test if the methods work. For this we decided to use the application called Postman. Postman is an application that allows the user to document their API, create simple HTTP requests and view the responses from them.

When we send the request, the app displays the success code and the return value if there's any. This was a simple way to test and debug our API. Every time we created a new request, we exported it to git repository, so everyone had the access latest postman requests.

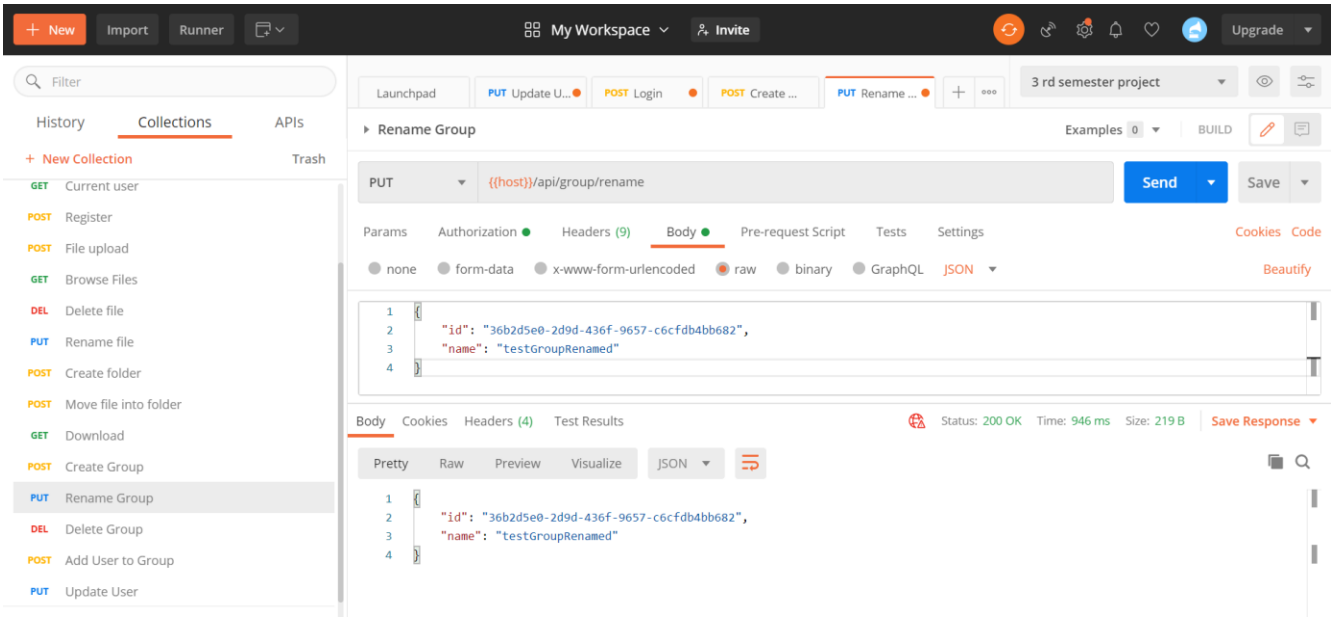


Figure 47: Sending a request and getting the results in Postman

Part VI. Conclusion

Lastly, we would like to reflect on the project we have been developing for the past five weeks, look back on our problem statement, comment on the group work and evaluate the project from our perspective.

1. Project Evaluation

1.1 Choice of the project, scope

We were surprised by how well we decided the scope of our project; since we could have chosen whatever we wanted; it was tough to find the right project to implement. We went for the file sharing, which had the “unknown” file handling as its core feature, but this has proven to be the best decision for us, since the difficulty of the project was challenging; at the beginning there were a lot of blind spots.

We have managed to design the scope of our system such that we had successfully implemented all but one user story from our product backlog and we have added only three more to it during the process. This clearly shows our understanding of the project from the very beginning.

1.2 Implementation of the project

The beginning was challenging because we decided to use the .NET Core and chose to implement the registration and login functionality as first user story. This took a lot of googling to implement correctly. After we had finished this, there was another challenge in front of us- Azure and its blob storage.

After resolving these two, we shared knowledge among each other and successfully continued our way to implement the solution. There were other challenges in the process, but those were resolved faster and with confidence as we became familiar with all the technologies.

To sum it up, the team is satisfied with the result. It has all the desired features and works well in majority of usages, since a lot of time was spent on (unfortunately only) manual testing of the application. There are further options and features to enrich our system with, but for the time period we were given, this is it.

1.3 Client prioritization

At the very beginning of our project the team expected both clients to have the same functionalities. As the project progressed, we found out that it is not feasible to keep both clients at the same state and that we have to prioritize one of them.

After a brief discussion, it was decided that the Web client will be the one with all the functionalities and that we will implement as many features into the desktop client as possible.

At the end, all the critical functionality is implemented in both our clients, the desktop one is missing a few features, but the web client is fully implemented.

2. Groupwork Evaluation

Since the group has been formed only before the project, the team members neither did know each other nor had any idea about the coding skills and capabilities of others. That is why the project start was messy, combined with the fact that our first user story dealt with things we did not know how to code also did not help the morale. This is where pair programming helped, we had multiple sessions of it where we researched ideas and browsed the internet until we found a solution.

After these few days, we started getting know each other, we knew what to expect, what to ask from who, how to do pair programming and everything.

Even though there were a lot of communication issues and the pace was not sufficient for the whole duration of the project, implementing of this project was joyful and we gained a lot of experience while coding it.

3. Lookback to PS questions

At the very beginning, we wrote our problem statement where we outlined questions we strived to answer positively at the end of the project:

Question 1: Are we able to version the files and to store their previous versions, so that the users could access them easily?

Answer: The team is more than happy to announce that this was successfully implemented; for now, user can see the versions of the given file in the right sidebar in the web client and can revert to whichever version he wants. A log with the name of the user that changed the file contents, and a timestamp is visible too.

Question 2: Are we able to prevent the user from losing the progress he made while editing or commenting on a document by either caching or autosaving?

Answer: This was the main problem with editing text files at the same time, where we implemented optimistic concurrency. When other user edits the file at the same time, after trying to save the current version, a pop-up alert would inform him, that the file was changed while he was editing it. It is then user's choice what version of the file he wants to keep.

Question 3: Are we able to resolve the concurrency issues (two users working on the same file at the same time) in the system?

Answer: Yes, we were able to identify a concurrency issue in our project and to resolve it properly.

Question 4: Will the data in our system be safe and protected against various attacks?

Answer: The team is confident about the security of the project. SQL injection attacks cannot occur at all, the passwords are hashed and salted in the database, CRUD permissions in groups are handled smoothly and it is group admin's choice who will be able to perform which operations.

References

[1] Karthick, "Google drive – 18 Amazing Stats and Facts," HelloLeads, 1 July 2020. [Online]. Available: https://www.helloleads.io/blog/stats-facts/google-drive-18-amazing-stats-and-facts/. [Accessed 19 December 2020].

[2] N. Galov, "25 Must-Know Cloud Computing Statistics in 2020," Hosting Tribunal, 4 December 2020. [Online]. Available: https://hostingtribunal.com/blog/cloud-computing-statistics/#gref. [Accessed 19 December 2020].

[3] MaterialDesignInXamlToolkit, "Material Design In XAML Toolkit," Google, 21 September 2020. [Online]. Available: https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit. [Accessed 17 December 2020].

[4] Newtonsoft, "Json.NET," Newtonsoft, 9 November 2019. [Online]. Available: https://www.newtonsoft.com/json. [Accessed 27 November 2020].

[5] N. Tomar, "WPF: An Introduction," C# Corner, 4 September 2018. [Online]. Available: https://www.c-sharpcorner.com/uploadfile/nipuntomar/wpf-an-introduction-part-1/. [Accessed 18 December 2020].

[6] Wikipedia, "Data binding," Wikipedia, 17 December 2020. [Online]. Available: https://en.wikipedia.org/wiki/Data_binding. [Accessed 20 December 2020].

[7] Microsoft, "INotifyPropertyChanged.PropertyChanged Event," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.inotifypropertychanged.propertychanged?view=net-5.0. [Accessed 12 December 2020].

[8] Bootstrap, "Modal components," Twitter, [Online]. Available: https://getbootstrap.com/docs/4.5/components/modal/#examples. [Accessed 6 December 2020].

[9] W3Schools, "SQL Injection," Refsnes Data, [Online]. Available: https://www.w3schools.com/sql/sql_injection.asp. [Accessed 16 December 2020].

[10] I. Sommerville, Software Engineering, Global 10th ed., Pearson, 2015.

[11] H. Kniberg and M. Skarin, "Kanban and Scrum - Making the Most of Both," InfoQ, 21 December 2009. [Online]. Available: https://www.infoq.com/minibooks/kanban-scrum-minibook/. [Accessed 19 December 2020].

2] P. Kemp and P. Smith, "Waterfall Model of System Development," Wikipedia, 14 June 2010. [Online]. Available: https://en.wikipedia.org/wiki/Waterfall_model#/media/File:Waterfall_model.svg. [Accessed 19 December 2020].

[13] M. H. Iversen, "Introduction to SCRUM," 7 September 2020. [Online]. Available: https://ucn.instructure.com/courses/22014/files/folder/modules/Module%203%20XP%20and%20Scrum. [Accessed 19 December 2020].

[14] Dutchguilder, "RUP phases and disciplines," 16 October 2007. [Online]. Available: https://en.wikipedia.org/wiki/Rational_Unified_Process#/media/File:Development-iterative.png. [Accessed 19 December 2020].

[15] M. Cohn, User Stories Applied, For Agile Software Development, Repr. ed., New Jersey: Pearson Education (US), 2002, p. 166; 169.

[16] K. Beck, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland and Thomas Dave, "Manifesto for Agile Software Development," 2001. [Online]. Available: https://agilemanifesto.org/iso/en/manifesto.html. [Accessed 19 December 2020].

[17] Visual Paradigm, "What is Planning Poker in Agile?," Visual Paradigm, [Online]. Available: https://www.visual-paradigm.com/scrum/what-is-agile-planning-poker/. [Accessed 19 December 2020].

[18] Guru99, "MVC vs MVVM: Key Differences with Examples," Guru99, [Online]. Available: https://www.guru99.com/mvc-vs-mvvm.html. [Accessed 19 December 2020].

Appendices

Appendix A: Problem Statement

Problem	The need of a simplistic file sharing application
Group members	Dominik Ábel Sári, Erik Petra, Győző Csuhai, Krisztián Henrik Papp, Marek Strúcka
Brief problem description	The issue is that there is no easy-to-use group collaboration software aimed at file sharing.
Problem	All creative workflows, regardless of the field, technical aspects or level of expertise, rely on communication and collaboration between individuals and groups of people. Our aim is to provide a file-sharing system offering easy and efficient sharing, versioning and commenting of documents and other files, to ease collaborative efforts within projects involving multiple persons. The system will allow users to: • Quickly preview files • Pin and attach notes and comments to specific parts of documents • Allow sharing files between groups of people or individuals
Problem statement	Are we able to design a file-sharing application that would make the process of file-sharing among groups more effective? • Are we able to version the files and to store their previous versions, so that the users could access them easily? • Are we able to ensure the smoothness, faultlessness, and user-friendliness of the new system? • How well can we follow the guidelines of agile software development

	methodology? How well will it work for us? • Are we able to prevent the user from losing the progress he made while editing or commenting on a document by either caching or autosaving? • Are we able to resolve the concurrency issues (two users working on the same file at the same time) in the system? • Will the data in our system be safe and protected against various attacks?
Technology used	The .NET Core framework will be our main development tool. Our application will have two clients (an ASP.NET MVC web client and a Windows Forms desktop client) which will connect and get information from a RESTful API service (based on ASP.NET Web API + MVC for the documentation of the API). For version control, we plan to use Git and host our repositories on GitHub. For databases, Microsoft SQL Server hosted and managed by Azure in the form of an Azure SQL Database.

Appendix B: Group Contract

Everyone signed below agrees to the terms that were discussed and agreed upon on the very first day of the project:

Everyone will participate fully in the project for the whole duration of it

Everyone will be present at daily scrum meetings that will be held each day at 8:30

When someone cannot make it to the meeting, he will inform the group in advance and make sure to finish his tasks remotely

When someone cannot finish his tasks on time, he will immediately inform the group and try to resolve the problem firstly on his own, they among the group members

When someone disagrees with the presented idea, he will start a discussion about the problem.

When there is a dispute, a vote is held - since there are 5 people in the group, the option with 3 or more votes wins

When a member is not participating in the group work or the daily scrum meetings, he may be sacked from the group

Aalborg, 27.10.2020

.....	.....	.....	.....	.....
Dominik	Erik	Győző	Krisztián	Marek