BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER-SCIENCE, ENGLISH

DIPLOMA THESIS

Web Based Collaborative Content Viewing

Supervisor
Assoc. Prof. Rareș Florin Boian, PhD

Author
Ciorbă Rareş-Nicolaie

2022

ABSTRACT

This thesis explores the problem that other web-based collaborative content viewing applications have: the simplicity of their permission system. In order to do this, the problem is thoroughly explained with real life examples in Chapter 2.

However the reason why most other applications have the problem of having a permission system that is too simple is because implementing an exhaustive permission system is a massive task and requires a lot of infrastructure and work. In order to do this properly, in Chapter 3, the thesis goes over different kind of access control mechanisms and tries to choose the best one to implement.

Chapter 4 explores the application from beginning to end. An overview of the requirements is present as well as motivation behind picking all of the technologies that have been picked. After that the software requirements are explained as well as how the project is setup. Lastly, the implementation details and the usage of the application is explained.

Finally, the thesis ends with the Chapter 5, the conclusion and future work chapter, in which what has been learned from the thesis is explained as well as how the application could be further expanded.

Contents

Chapter 1

Introduction

1.1 What is a permission system?

To explain what a permission system is, firstly a need to understand what exactly a permission is exists. A permission is more or less a rule which is applied to either an object or an action. A permission system is a way of handling and managing the permissions, as well as putting them into use.

1.2 Why are permission systems necessary?

A permission system is necessary because it let's a user manage individual or group access and handle constraints around various objects, actions, users, or roles. In essence, a permission system has been in use for a very long time. The best example regarding a permission system is a file permission system.

1.3 What is the purpose of this thesis?

The purpose of this thesis is to provide an interactive and more exhaustive system of control in a web environment. The problem that this thesis is trying to address is that most collaborative content viewing solutions such as: Google Drive, or Dropbox explore very simplistic permissions. But because of their simplistic permission approach, a need for a very specific structure for files and folder is needed in other to facilitate a complete control over all of the data a user wants to share.

Because the needs of a user and what they would want to share with other users will always change, having a very specific structure in mind for the files and folders becomes a big problem that becomes very chaotic and hard to manage. In order to solve this issue, this thesis proposes a more exhaustive permission system for the content of the user, that allows a lot more control.

1.4 Thesis structure

The thesis will contain an additional 5 chapters.

- Permission implementation in other apps — This chapter explores what other applications have done regarding their permission implementation, and why a more exhaustive permission control system might have advantages over the simplistic approaches of other applications.
- Permission management — How exactly are permissions handled and what are the different ways of managing permissions by exploring two major attribute control systems — ABAC and RBAC.
- The application — This chapter explores how the application actually manages and solves the problem proposed in this thesis. What technologies it uses, and other technical details revolving around the app.
- Conclusions and Future work

Chapter 2

Permission implementations in other applications

As previously mentioned, most implementations for permissions in other web based collaborative content viewing applications are too simplistic and don't provide the user with a complete control over the actions another user takes or the influence they can have on objects.

2.1 Google Drive

One of the most popular systems for sharing files between users is Google Drive. Google Drive introduces three ways of sharing a file or a folder with other users: sharing via contact, sharing via a link, or publishing.

However, associated with these three types of sharing files or folders, a user can set three types of permissions on the files they wish to share: can edit, can view, or can comment. [1]

The "can edit" permission means that the user that gets access to the resource will be able to edit the content of the resource in its entirety.

The "can view" permission is just a read-only permission, thus the user is not allowed to modify the file or folder they have access to.

The "can comment" permission means that the user gets access to Google Drive's comment section and can leave comments regarding the contents of a document.

Although this system is very simple to use it comes with a few drawbacks. There are a few scenarios where sharing documents with the proper permission might cause be an issue.

Consider the file structure as shown in figure 2.1, assume that the user wants to give access to the folder called "Data" to another user. Currently if the user wants to protect the folder called "Sensitive Data", the only way to do that would be to

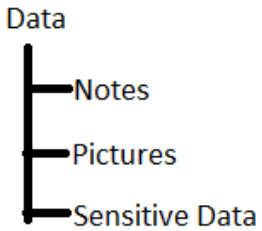

Figure 2.1: Hard to express file structure

to share ”Notes” and ”Pictures” individually without exposing the whole ”Data” folder. For this example doing that might work, but once we add a lot of other folders, the whole situation becomes very tedious.

2.2 Unix systems

Although not an online system, unix systems have a very good way of dealing with permissions which is worth going over. The core premise of the unix permission system is that permissions are divided into 3 actions: read, write, execute, along 3 subjects, the owner, the group and other users.

Alongside all of these, unix systems also provide a so-called ”god” role, the root role, which has access over every file and folder, and they can freely execute any command, which involves tinkering with the permissions of other users, groups, files, or folders. In actuality the ”root” user is one of the most targeted place of attack, because if a malicious person gains access to the root account, they can do anything in the system.

The problem unix system have with their permission is, again, the lack of granularity. Even though they provide access for the owner, the group, and other users, the fact that you can only control access to one owner, and to one group can lead to issues.

Let’s take the file structure expressed in 2.2 as an example. The system has 3 roles: a role for the user — the owner of the system — and 2 other roles representing services that are running on the system. Due to security reasons, the services run in their own separate roles.

Taking the permissions as described in 2.1, the first role, the user wants complete access ove the external hard drive. The second role, ”service1” wants access to just the ”Music” folder, without knowing about the existence of the ”Movies” folder. And finally the third role ”service2” wants access to the ”Movies” folder without knowing about the existence of the ”Music” folder.

With just these three roles, the permission scheme proposed is easily achievable as described in 2.2.

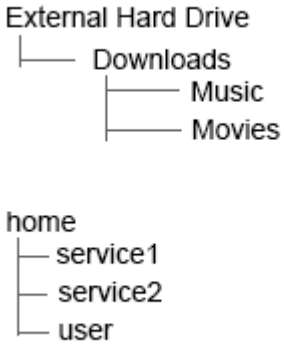

Figure 2.2: Complex file structure

User name	Access	Description
user	Everything from the External Hard Drive	The user of the system
service1	Music	-
service2	Movies	-

Table 2.1: User roles and permissions

Folder	Owner	Group	Other
External HDD	user - rw	user - rw	-
Music	user - rw	service1 - rw	-
Movies	user - rw	service2 - rw	-

Table 2.2: User roles and permissions

The first column represents the folder. The second column represents who the owner is, and the ”rw” represents that they have read write access. The third column represents what the group is, and the ”rw”, again represents that they have read write access. For the sake of example and simplicity the ”execute” permission will be left out. The other column represents what permissions other users have, i.e the ones that aren't the owner or are part of the group.

With this permission schema, the user, being the owner of the ”External HDD” folder has access over it and all of its subfolders. The role ”service1” has access to the ”Music” folder, but because the ”Music” folder has no read or write permission specified for ”others” that means that it can not be accessed by ”service2”. The same idea was applied to the ”Movies” folder, and for the role ”service2”.

Even though this is a fairly simple example, the structure of the permission system is a bit cluttered and hard to manage, but in the end it does represent the file structure aforementioned.

The problem with this rises once we introduce more users and more folders.

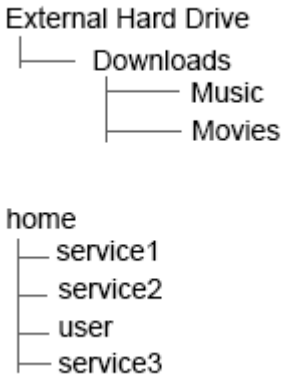

Figure 2.3: Example bad scenario for unix

As shown in figure 2.3, the file structure remains the same but a single role has been added. In this case the role ”service3” can be seen as a file sharing service such as sambashare. The role ”service3” requires complete access over the whole directory that is ”Downloads”, but no other directories on the External Hard Drive.

Due to the fact that the unix system provides control over a single role, a single owner, and all other users, there is no way of achieving this file structure, and this permission structure, without having to change the structure of the files.

Chapter 3

Permission management

In order for the application to provide the necessary security over files a permission system needs to be implemented. There are various ways of implementing permissions, two fairly used techniques are Attribute-based access control (ABAC) and Role-based access control (RBAC).

3.1 Attribute-based access control

Attribute-based access control (ABAC) is an authorization model that evaluates attributes (or characteristics), rather than roles, to determine access.[2] The purpose of this attribute-based authorization model is to preserve the security and integrity of data or devices by restricting access only to authorized users.

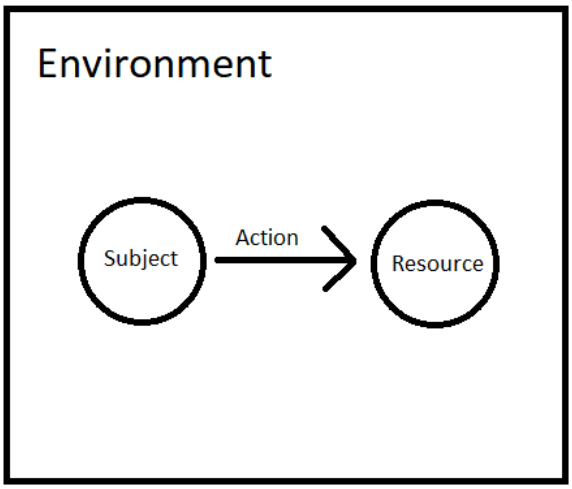

Figure 3.1: The 4 major components of ABAC

ABAC contains 4 major components: Subject, Resource, Action, and Environment as seen in 3.1.

Although the subject of ABAC is usually associated with a user, in reality, the Subject can be either a role, a user profile, or essentially any identifying criteria. Generally, the Subject is identified by their authentication token, for example even something like the Personal Numeric Code can be used.

The Resource is an asset or an object — be it a file, an application, a server or an API — that the Subject wants to access.

The Action represents what the Subject wants to perform on the Resource. A good example for the Actions is the common actions a file explorer can perform: read, write, edit, copy, delete. An Action can have multiple attributes (parameters) associated with it. For example, a copy could have the source and the destination.

The Environment represents the context in which the Action has been performed. The context can contain data such as: the date and the time when the action was performed, the location from which the Subject is trying to perform the Action, or even the communication protocol or information about the encryption strength.

Attributes are the characteristics involved during an access event. Attribute-based access control analyzes these attributes against a set of rules in order to determine if the subject is authorized.

Advantages

The key benefit of ABAC is its flexibility. Essentially, the limit for policy-making lies in what attributes must be accounted for, and the conditions the computational language can express. ABAC allows for the greatest breadth of subjects to access the greatest amount of resources without requiring admins to specify relationships between each subject and object. Take the following steps as an example:

- When a subject joins an organization, they're assigned a set of subject attributes (e.g., John Doe is a consultant for the radiology department).
- An object, when created, is assigned its attributes (e.g., a folder with cardiac imaging test files for heart patients).
- The admin or object owner then creates an access control rule (e.g., "All consultants for the radiology department can view and share cardiac imaging test files for heart patients").

Admins can further modify these attributes and access control rules to fit the needs of an organization. For instance, when defining new access policies for external subjects like contractors and providers, they can do so without manually changing each subject-object relationship. ABAC allows for a wide variety of access situations with little administrative oversight.

Disadvantages

ABAC can be difficult to get off the ground. Admins need to manually define attributes, assign them to every component, and create a central policy engine that determines what attributes are allowed to do, based on various conditions ("if X, then Y"). The model's focus on attributes also makes it hard to gauge the permissions available to specific users before all attributes and rules are in place.[2]

However, while implementing ABAC can take considerable time and resources, the effort does pay off. Admins can copy and reuse attributes for similar components and user positions, and ABAC's adaptability means that maintaining policies for new users and access situations is a relatively "hands-off" affair.

3.2 Role-based access control

Role-based access control (RBAC) is an authorization model where the roles are the only ways to determine the access to a certain resource or action. A certain simplicity in the ABAC idea is appealing. If a user has attributes that are reflected in the objects they want to access, then access is granted. On the other hand, with RBAC, the permissions granted to a user through roles must be evaluated to determine if the desired access will be granted.[3]

Advantages

Unlike ABAC, where each individual users has a set of attributes, with RBAC all users are assigned a role, which makes the management of the permissions for a set of users easier to manage. Other key benefits of RBAC include:

- The possibility of creating a re-assignable sets of permissions.
- Reduce the risk of human error when assigning permissions.
- Granularity of permissions is still preserved by breaking up permissions into multiple roles and assigning multiple roles to users.
- Reduce administrative work by having the possibility of quickly changing the user's roles (i.e after a project ends and an user access needs to be revoked).
- Decrease the security risks by restricting access to sensitive information in a more controlled manner.

Disadvantages

Despite having a simpler implementation compared to ABAC, RBAC does of course come with disadvantages. The main disadvantage of RBAC is that due to the increasing number of different roles, managing said roles becomes a very complex task if the management wishes to have properly encapsulated permissions.

Granularity is also much harder to achieve than with an ABAC system. In the case where you want only a user to have a privilege and that user is part of a group already, the only way to give that user and only that user the privilege is to create a new role solely for that.

Chapter 4

The application

4.1 Overview of the requirements

As mentioned in the 2nd chapter of the thesis, one of the major problems in many applications that revolve around permissions is the fact that they don't provide a granular enough system. In order to deal with this, unlike the Google Drive, the demo application must implement an exhaustive system, and unlike unix systems, the application must allow multiple roles to be assignable to a single user.

In order to achieve this the application must have the following features:

- A user system, and therefore the ability to switch between the perspective of different users.
- A way of managing roles: assigning roles to users, changing permissions of the roles, adding/removing roles.
- And finally, a way of actually testing the permissions out in a realistic environment.

4.2 Backend — used technologies

4.2.1 .NET and C#

.NET is a free, cross-platform and open source framework developed by Microsoft, that started as proprietary. The reason .NET was chosen for this application is because it provides an environment for building very expandable Rest APIs.

One of the key benefits of .NET core is Entity Framework Core. Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.

EF Core can serve as an object-relational mapper (O/RM), which[4]:

- Enables .NET developers to work with a database using .NET objects.
- Eliminates the need for most of the data-access code that typically needs to be written.

EF Core supports many database engines such as SQLite, PostgreSQL, MySQL, etc.

4.2.2 LINQ

LINQ or Language integrated query is the name for a set of technologies based on integrating SQL-like queries straight into C#, which has the benefit of compile-time checking rather than runtime. Writing SQL code as a set of string is very error prone and fairly hard to manage, and LINQ helps to alleviate that problem.

To do that, LINQ provides SQL-specific keywords such as `from`, `where`, `select` which can be used on collections as shown in figure 4.1.

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Figure 4.1: LINQ Example

The LINQ system further contributed to the choice of using .NET for the application, since there were a lot of interconnected entities.

4.2.3 JSON Web Token

Even though the scope of this thesis is to explore the permission management in an application focused on collaborative content viewing, it would all be futile if the application is easily exploitable through security vulnerabilities, and thus a need for a secure and encrypted means of communication between the front end and the back end exists.

The JSON Web Token or JWT for short is a proposed internet standard for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims.

4.3 Frontend — used technologies

4.3.1 React.js

React (also known as React.js or ReactJS) is a free and open-source front-end JavaScript library for building user interfaces based on UI components. It is maintained by Meta (formerly Facebook) and a community of individual developers and companies.

React is component based which allows the user to build encapsulated components that manage their own state, and then further compose them into a more complex user interface.

React also makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes. [5]

Declarative views make your code more predictable and easier to debug.

4.3.2 Typescript

TypeScript is a programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language. It is designed for the development of large applications and transpiles to JavaScript. As it is a superset of JavaScript, existing JavaScript programs are also valid TypeScript programs.

TypeScript provides static typing through type annotations to enable type checking at compile time. This is optional and can be ignored to use the regular dynamic typing of JavaScript.

Because Typescript provides typing, developing becomes overall easier as code is much easier to debug since you won't rely on runtime errors to spot bugs, but instead, the code fails at compile time.

4.3.3 Material UI

The main reason react was chosen lies within the material ui library. MUI is a ReactJS library that provides components based on the Material UI standard developed by Google. It allows a quick and easy way of developing responsive applications without the need of worrying about the aesthetic of the application.

The components that it provides also come with a lot of functionality by default which further saves time during development.

4.4 Software Requirements

In order to run both the projects, an operating system is needed. Due to the fact that the backend project uses .NET, the software requirements regarding the operating system are mostly given by it. Hence a machine with an operating system that can run dotnet CLI at minimum is needed.

Known supported operating systems include[6]:

- Windows and Windows server
- MacOS
- Linux and linux servers: Ubuntu, Alpine, CentOS, Debian, etc.

For the frontend the major dependency is React.js, in order to install that Node.js is needed.

The operating system requirements for nodejs are pretty much the same as the ones for dotnet[7]:

- Windows
- MacOS
- Linux: Ubuntu, Debian, CentOS, etc.

Since both projects are essentially source code that needs to be compiled, an IDE (Integrated Development Environment) is strongly recommended for both of them.

For the backend project there are many different choices: Rider, Visual Studio, Visual Studio Code. As for the frontend project, any IDE that has support for html, css, typescript should suffice, for example InteliJ, PhpStorm, WebStorm, Visual Studio Code, Sublime Text.

During the development phase, I used Rider for the backend and PhpStorm for the front end, which I strongly recommend as that is what will be used to explain the setup and running of the projects.

4.5 Setup

Since the application is split into two different projects, each one of them needs to be set up individually.

4.5.1 Backend

The backend is written in C# with .NET framework core. The first requirement is to have dotnet installed. After that the app settings need to be configured. To do that, navigate to `Backend/Backend`, and copy paste the template file called `appsettings.template.json` into another file called `appsettings.json`.

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "DataContext": "TODO"
    },
    "JWT": {
        "ValidAudience": "TODO",
        "ValidIssuer": "TODO",
        "Secret": "TODO"
    },
    "Admin": {
        "Email": "admin@mail.com",
        "Password": "<your very secret password>"
    },
    "RootPathPoints": [
        "Path/To/File"
    ]
}
```

Figure 4.2: App settings template

Next, as shown in figure 4.2, a few fields need to be filled. The first field is related to the `ConnectionStrings`. A ConnectionString is a way of specifying how to connect the application to the database. During development, the database used was SQLite.

To specify a connection to that set `DataContext` equal to `"Data Source=<Location>;"`, where Location will be the location of the database.

Example: `"DataContext": "Data Source=Kumo.db;"`.

The second field that needs to be configured is related to the JSON Web Token (JWT). The ValidAudience represents the location at which the requests with the token will be valid. For example if you want to have only one computer use the API, then the IP address of that computer will become the ValidAudience, meanwhile in

the case where anybody can be a valid audience, the field is set to `*`. Next, the ValidIssuer represents the location from which the token can be created. This is always the address associated with the server.

The final and the most important field from the JWT configuration is the `Secret`. This represents a base64 encoded string which is used for creating the token. If this field is not secure enough, or it gets leaked, then the JWT becomes reversible, which is why this field is extremely important to remain a secret.

The third field that needs to be configured is related to the Admin. Normally having an endpoint that creates an administrator user is bad practice, which is why the application handles this by providing a way of creating the administrator user at the first start of the application. The `Email` and `Password` fields are explanatory.

The fourth field is related to the Root Path Points. In short, a root path point is a place from which the navigation can start. This field is not mandatory as these can be created from the front end, via the administrator account.

After setting up the `appsettings.json` file, the next step is to create the database and all of the schema for it. To do that, we need to run migrations which require a dependency called `dotnet-ef`.

To install `dotnet-ef`, a few commands need to be run. The first one being `dotnet tool install --global dotnet-ef`. After that we can create the database, to do that you need to be in the same directory as the `appsettings.json` file, and then run `dotnet ef database update`.

With that, all the setting up for the back end is finished, and the backend is now runnable.

Open your IDE of choice, in this case Rider will be used. Simply open the project in it, and hit the `Run` button in the top right corner.

After that, you should see Swagger, like shown in the figure 4.3.

4.5.2 Frontend

Compared to the backend project, setting up the frontend is significantly easier, as only two commands need to be run. After navigating to the `frontend` directory, we need to install the dependencies. To do that simply open a terminal and write `npm install`.

After that completes, open your IDE of choice, in this case PhpStorm will be used. Open the project in it and then run it via the `Run` button.

Once the application has finished compiling, a browser will open and you will see a login screen, as shown in 4.4.

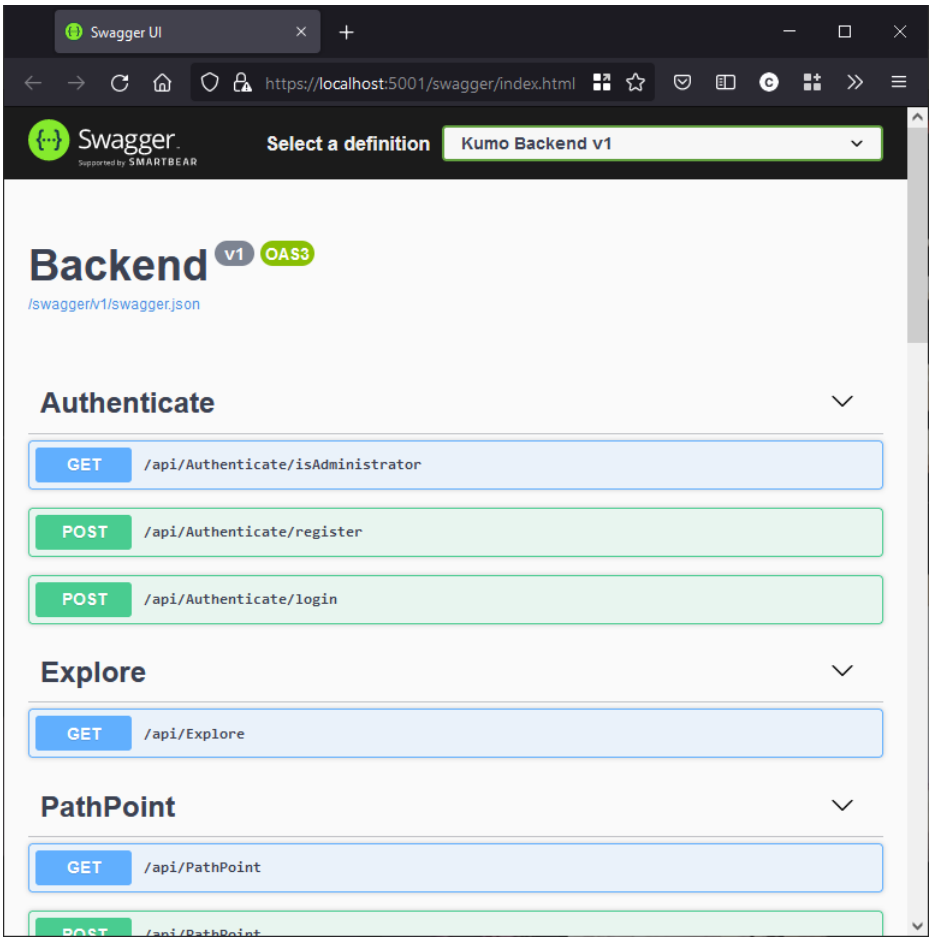

Figure 4.3: Swagger and the API endpoints

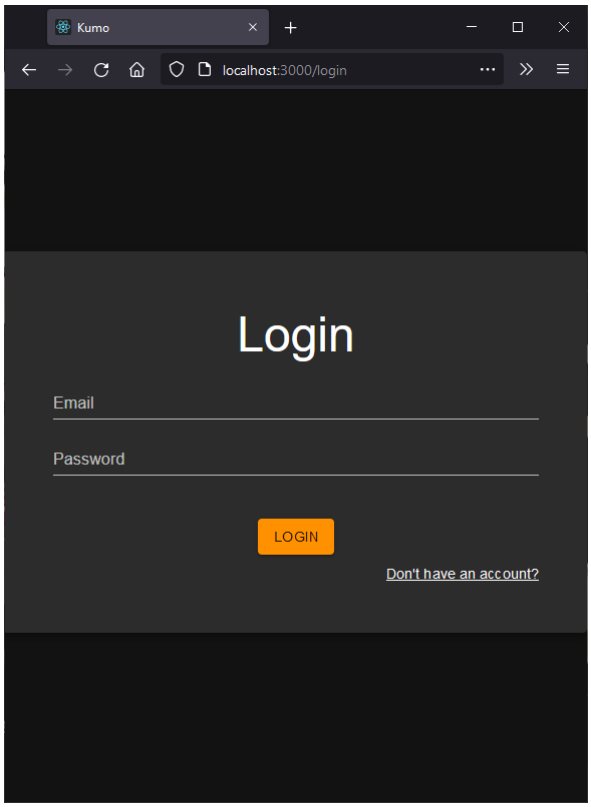

Figure 4.4: Frontend login page

4.6 Implementation details

The implementation of the application uses RBAC, hence a way of managing roles is needed. This however starts with correctly separating the data in the database to make different operations possible and to account for different use cases.

To do this the data is structures into 6 classes:

- PathPoint — represents a specific location, has one important property: `isRoot`. `isRoot` represents if the path should be displayed on the explorer's home page;
- FileSystemEntryType — an enum that specifies if the system entry type is a file or a folder, or unknown;
- Role — self explanatory, it represents a role;
- Permission — this is an n:m relation between a PathPoint and a Role, with additional data such as: Read, Write, Delete;
- User — this represents a user in the application;
- UserRole — this is an n:m relation between a User and a Role.

4.7 Application usage

As mentioned in the requirements, in order to explore the problem of permissions, 3 main functionalities are needed. A way of managing users, a way of managing permissions, and a real life usage of those permission.

4.7.1 Users

Thus, the first requirement the application must have is a way of handling users. The application has functionality for register4.5, login4.4, and logout marked with a (1) in 4.6.

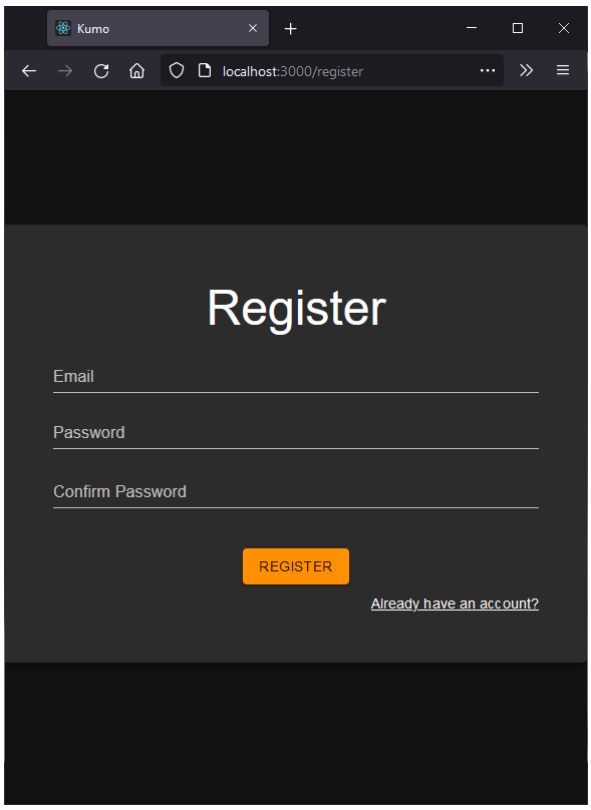

Figure 4.5: Frontend register page

Being trivial functionality, there is no reason to go over the flow of each of the actions.

4.7.2 Explorer

The real life usage of the permissions is an Explorer. The explorer, essentially, provides a way of navigating folders on a hard drive.

Being used as a proof of concept, the main purpose of the explorer is to give a visual representation of how the ”Read” permission works across all instances

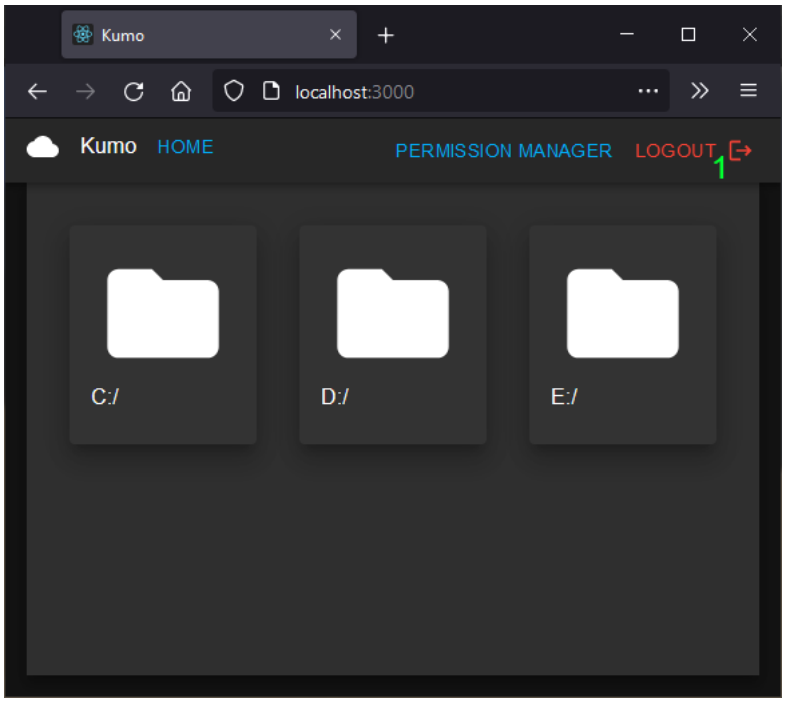

Figure 4.6: Frontend logout functionality on the main page

where it has been set. Due to the fact that the permission system takes into account permissions such as "Delete" and "Write", the Explorer can be easily expanded to accommodate for those too, however for the sake of example the Read permission will be used as a visual representation, while the response from the API calls will contain more information.

As mentioned in the implementation details, on the explorer home page4.7 the root path points are shown. When a user clicks on a directory, the explorer moves inside that subdirectory and shows the files and folders inside it. When clicking on the `E:/Test` subfolder it's contents are shown as in 4.8.

4.7.3 Permission management

The most important part of the application is the permission management system. As mentioned in Setup of the Backend, an admin account is needed. To properly show the functionality of this, I'm going to use two browsers which are connected to two accounts. As shown in 4.9, the browser on the left is connected to an admin account, which shows the `PERMISSION MANAGER` button, while the browser on the right doesn't.

You'll notice that in figure 4.9, the user on the right does not see any path points. And that is because there are no permissions configured yet aside from the `E:/Test` path point.

To configure the permissions, the first step is to go to the Permission Manager.

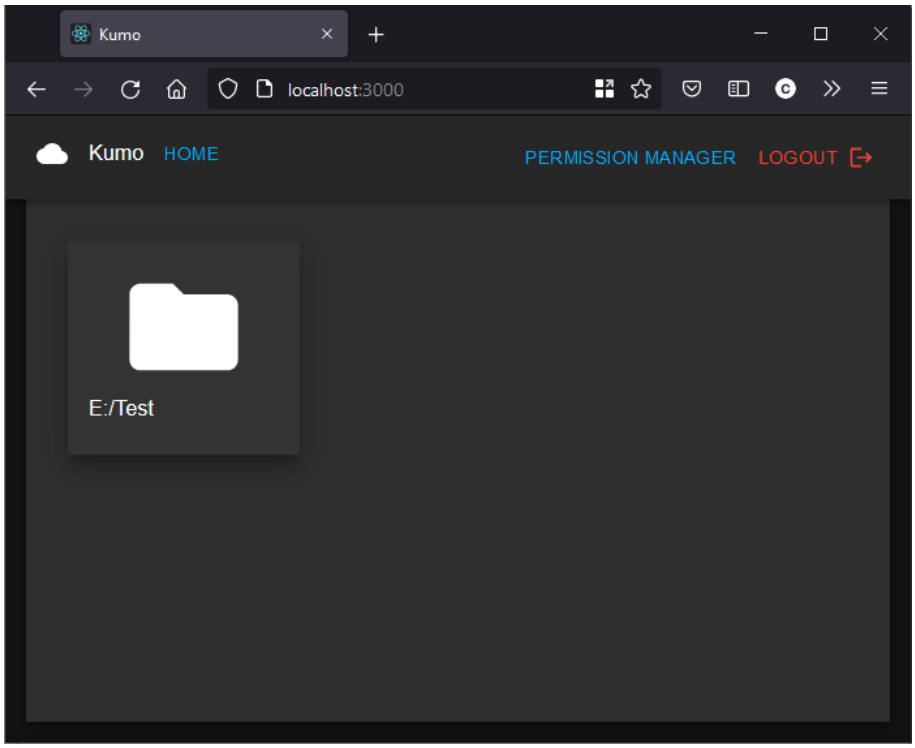

Figure 4.7: Explorer home path

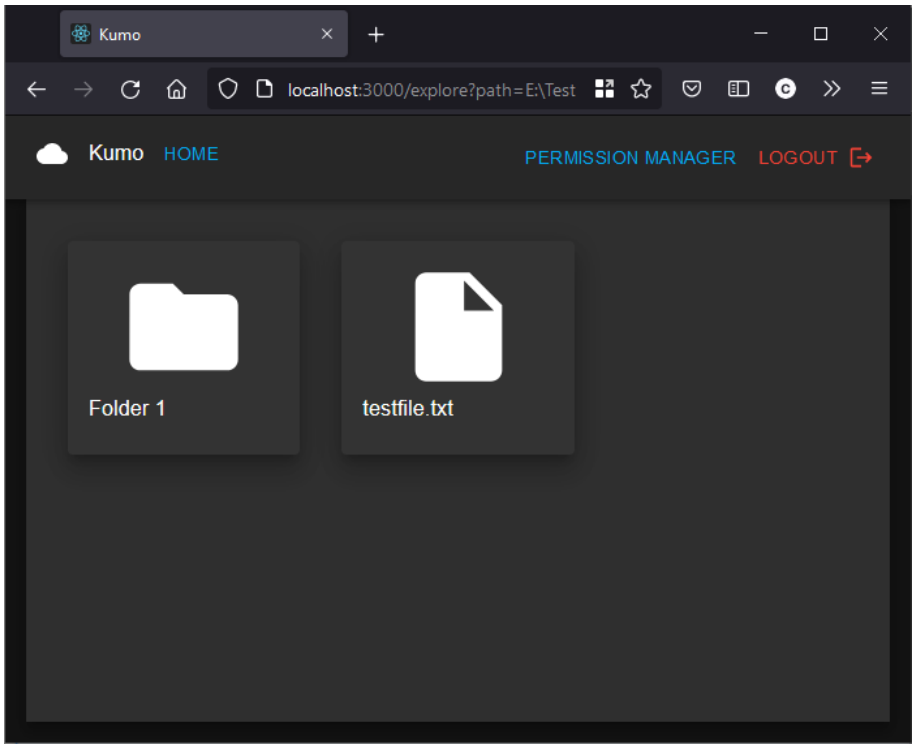

Figure 4.8: Explorer subfolder navigation

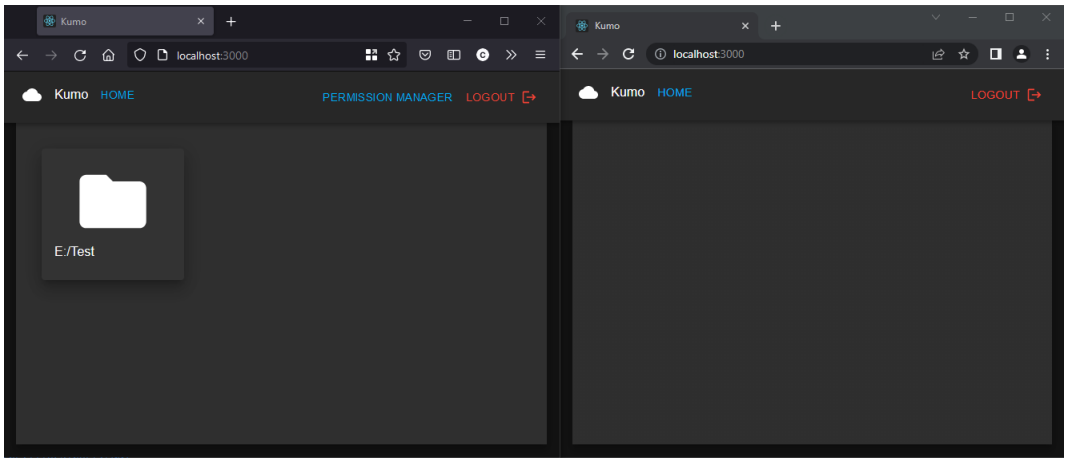

Figure 4.9: Permission manager button on an admin account and on an normal account

Due to the complexity of the permission manager, there are 4 configurable entries as shown in figure 4.10

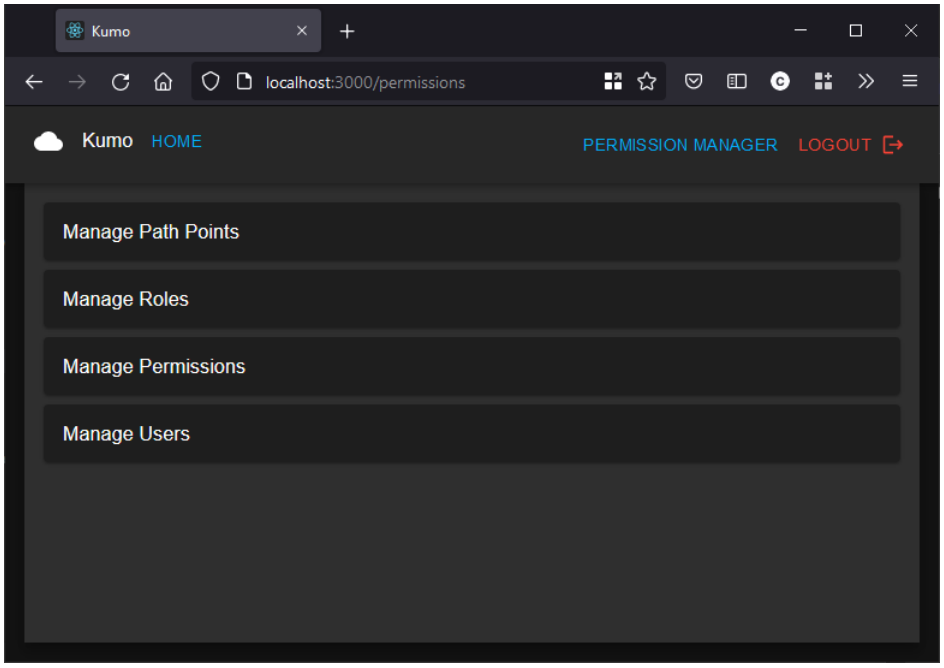

Figure 4.10: The default view of the permission manager

To demonstrate the power of the permission system, we will configure the scenario mentioned in chapter 2, the hard to express file structure from google drive shown in figure 2.1.

Configuring Path Points

Once you click on the `Manage Path Points` button, the accordion will expand and the path point manager will be shown as in figure 4.11

Firstly, all managers have a table that show the entries available in the database. In the case of the Path Point Manager, there are a number of important elements:

- (1) — Shows the path of the path point
- (2) — Shows whether or not the path point is root
- (3) — Marks the entry for deletion
- (4) — Commits the changes from the table to the database
- (5) — Cancels all of the changes and reverts to the original state from the database
- (6) — The path of the new path point to be added
- (7) — Whether or not the path point is a root path point
- (8) — Adds the path point to the database

Entries (1) and (2) can be edited by double clicking on them.

To achieve the file structure mentioned, where the Sensitive Data isn't accessible by a user, a path point that is not a root is needed for the `Sensitive Data` folder as shown in figure 4.12.

Configuring Roles

To achieve the desired permission structure only one role is needed, to do that configure it by opening the role manager as shown in 4.13.

The important elements in the role manager are as follows:

- (1) — The name of the role;
- (2) — Marks the element for deletion;
- (3) — Commits the changes from the table to the database;
- (4) — Cancels all of the changes and reverts to the original state from the database;
- (5) — The name of the role to be added;
- (6) — Adds the role to the database.

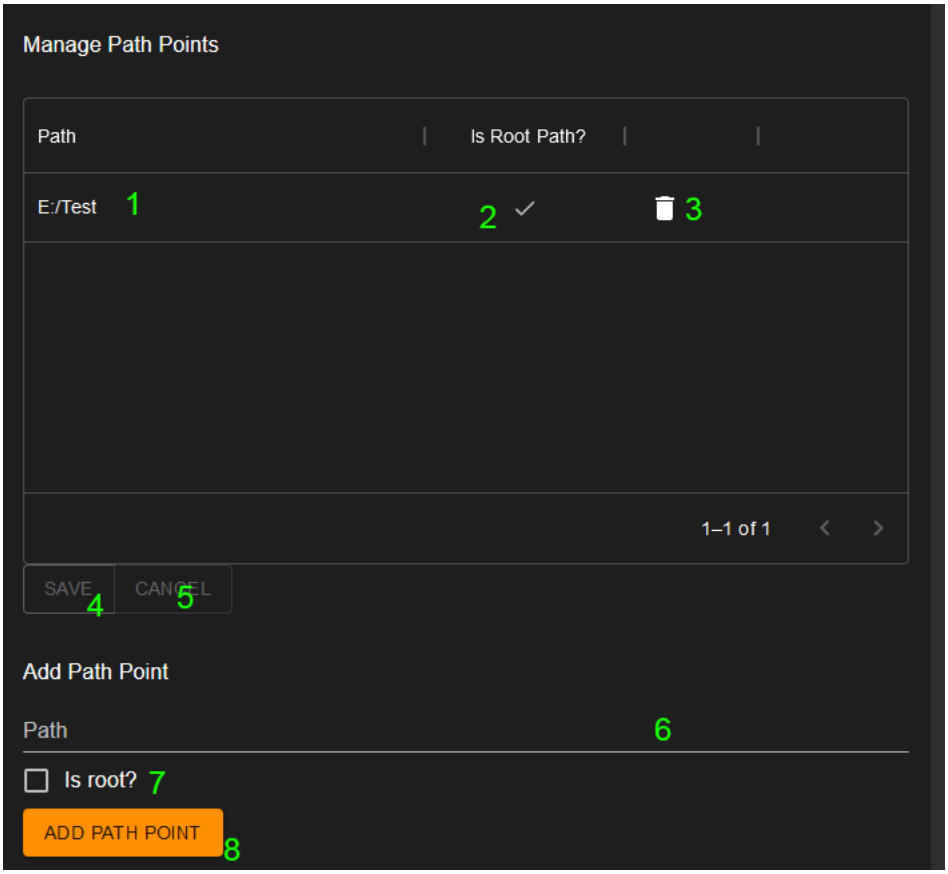

Figure 4.11: Path point manager

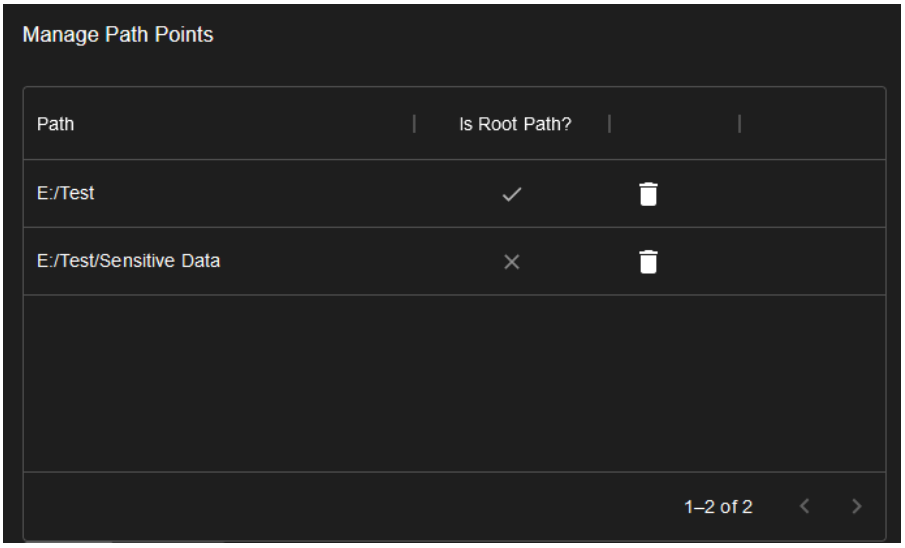

Figure 4.12: The correct path point structure needed

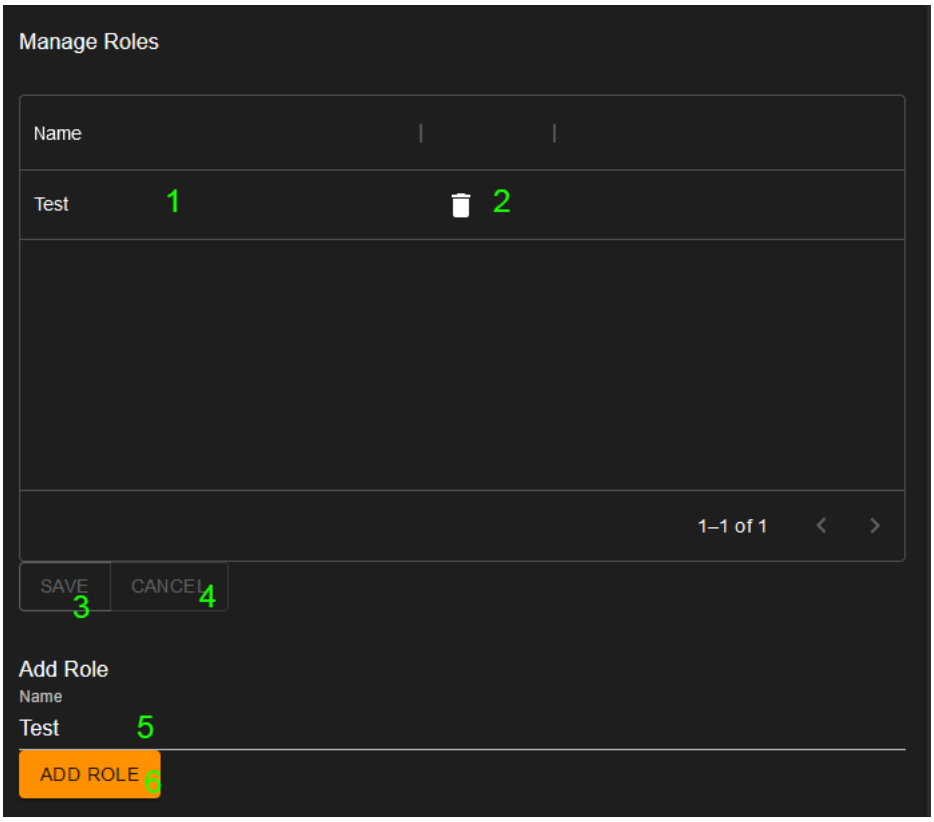

Figure 4.13: Roles manager

Configuring Permissions

For configuring permissions we essentially need to specify that the `E:/Test` path is readable by the `Test` role, and the `E:/Test/Sensitive Data` is not readable by the `Test` role.

As marked in figure 4.14, the important elements are as follows:

- (1) — The role for which the permission is applied;
- (2) — The path for which the permission is applied;
- (3), (4), (5), (6) — Whether or not the role can: Modify Root, Read, Write, or Delete;
- (7) — Marks the element for deletion;
- (8) — Commits the changes from the table to the database;
- (9) — Cancels all of the changes and reverts to the original state from the database;
- (10) — The role of the new permission
- (11) — The path point for the new permission

- (12), (13), (14), (15) — Whether or not the new role can: Modify Root, Read, Write, or Delete;
- (16) — Adds the permission to the database.

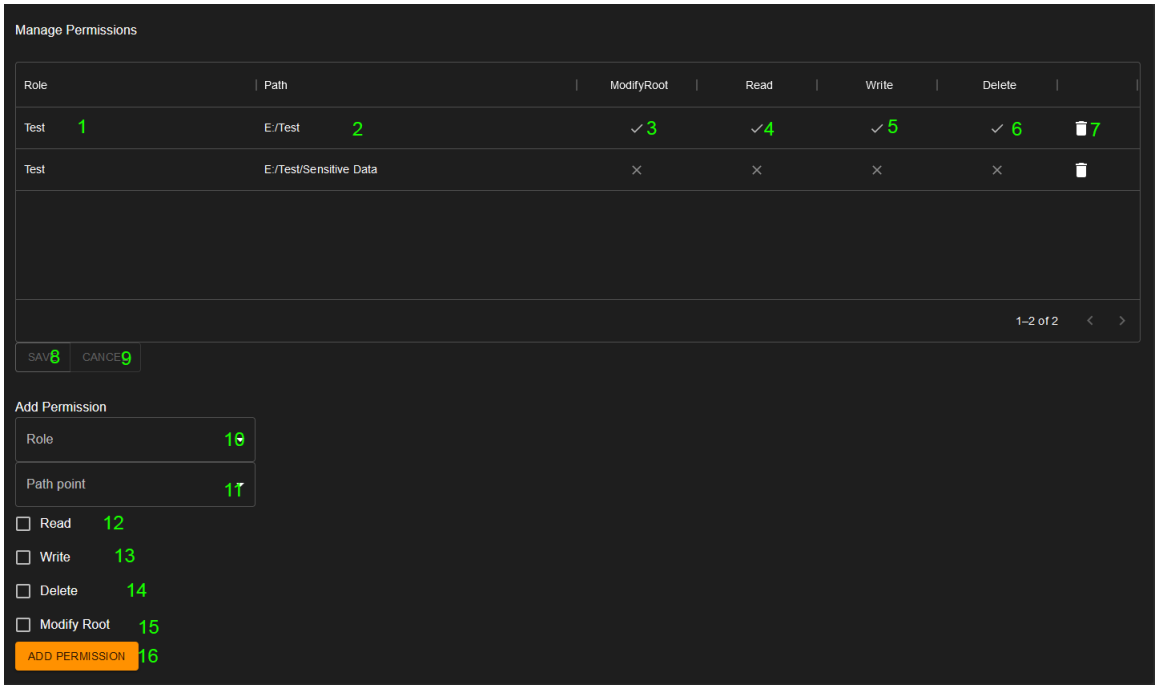

Figure 4.14: Permission manager

Configuring Users

The last step is to connect the `test@mail.com` user to the `Test` role.

As marked in figure 4.15, the important elements are as follows:

- (1) — The user that is linked;
- (2) — The role to which the user is linked;
- (3) — Marks the element for deletion;
- (4) — Commits the changes from the table to the database;
- (5) — Cancels all of the changes and reverts to the original state from the database;
- (6)— The user that will be linked;
- (7) — The role the user will be linked to;
- (8) — Links the user to the role in the database.

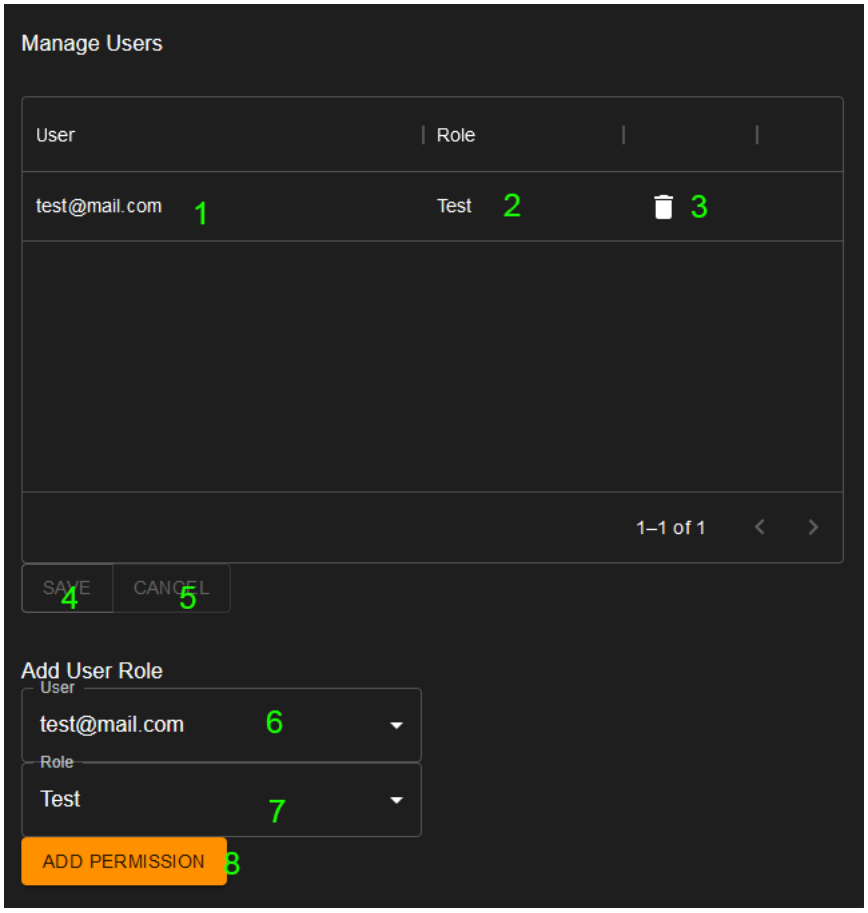

Figure 4.15: User manager

With that all of the configuration has been done, if the user `test@mail.com` navigates to the `E:/Test` path, they will be unable to see the `Sensitive Data` directory, as shown in figure 4.16.

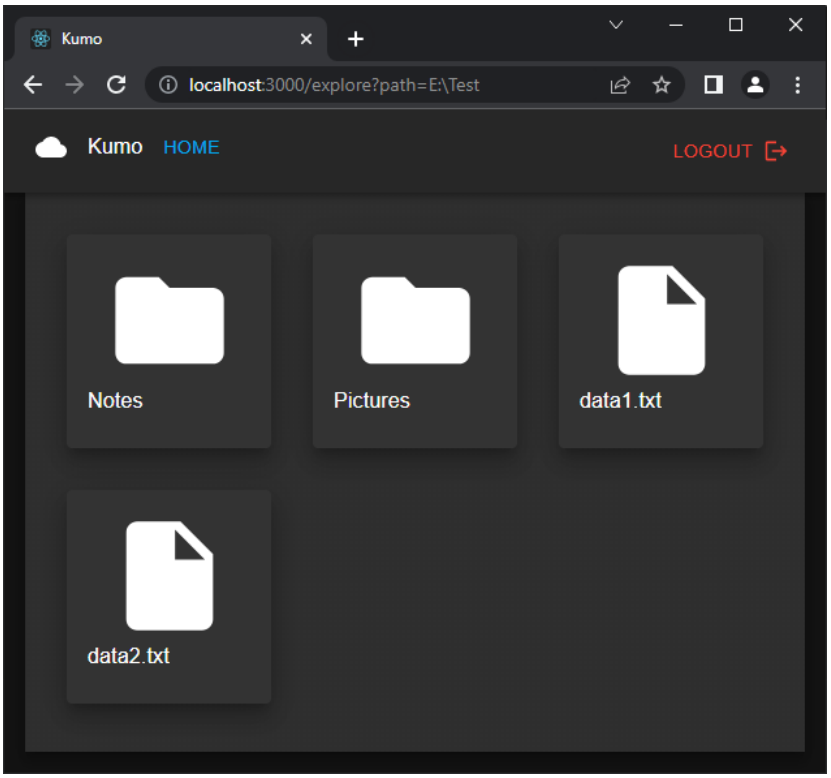

Figure 4.16: Permission manager result in the explorer

Chapter 5

Conclusions and Future Work

Conclusion

The complexity that was unachievable with Google Drive or Unix systems mention in Chapter 2 is completely achievable by implementing a Role-based attribute control system. Of course, with the increased complexity problems also arise. The ”Role explosion” problem mentioned in Chapter 3 remains an issue because in order to achieve proper granularity, a lot of roles will be needed.

Future work

With the application being a proof of concept, a lot of new features could be added to it. The explorer could be easily expanded by adding multiple views, more actions, and by overall offering more control over the navigation, files, and folders.

Due to the fact that the backend was written in C#, expanding the permission object is fairly easy as only a database migration is needed which is easily achievable with just a few commands.

Currently, there is only one role that can modify permissions, but allowing users to modify permissions in a hierarchy would also be possible. For example if a user has access to a specific directory, they can further modify the permissions of the subdirectories of that directory. But this of course comes with new challenges as the new system would become fairly cluttered and even harder to manage and to implement.

Bibliography

[1] Google Drive Support, "Set Drive users' sharing permissions." https://support.google.com/a/answer/60781?hl=en. Online; accessed 27 April 2022.

[2] Keith Casey, "What Is Attribute-Based Access Control (ABAC)?." https://www.okta.com/blog/2020/09/attribute-based-access-control-abac/. Online, Accessed: 10 March 2022.

[3] Ed Coyne, Timothy R. Weil, "ABAC and RBAC: Scalable, Flexible, and Auditable Access Management," 3 2013.

[4] Microsoft, "Entity Framework Core." https://docs.microsoft.com/en-us/ef/core/. Online; accessed 22 May 2022.

[5] Meta Platforms, "React." https://reactjs.org/. Online; accessed 22 May 2022.

[6] Microsoft, "Install .NET on Windows, Linux, and macOS." https://docs.microsoft.com/en-us/dotnet/core/install/. Online; accessed 21 May 2022.

[7] Contrast Security, "System requirements for the Node.js agent." https://docs.contrastsecurity.com/en/node-js-system-requirements.html. Online; accessed 21 May 2022.