Realizing Multi-Tenancy in Cloud Applications with several Organizational Environments on the example of PROCEED

by

Felipe Trost

Matriculation Number 456129

A thesis submitted to

Technische Universität Berlin
School IV - Electrical Engineering and Computer Science
Department of Telecommunication Systems
Service-centric Networking

Bachelor's Thesis

April 6, 2025

Supervised by:
Prof. Dr. Axel Küpper

Assistant supervisor:
Kai Grünert

Eidestattliche Erklärung / Statutory Declaration

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Berlin, April 6, 2025 Felipe Trost

Abstract

This thesis explores the integration of multi-tenancy functionality into cloud applications, on the example of the PROCEED Management System (MS). Currently, the PROCEED MS lacks comprehensive multi-tenancy support, restricting collaborative workflows primarily to individual users. To address this limitation, isolated environments were introduced, enabling multiple users or organizations to collaboratively manage their assets within distinct, secure workspaces.

A hierarchical folder structure was designed to mirror organizational structures and enhance asset management. The existing Role-Based Access Control (RBAC) system within PROCEED was also extended to incorporate environment-specific and folder-specific permissions.

Key aspects of the implementation included restructuring asset and user management, and establishing clear, efficient database schemas for environments and their related assets. The developed solution supports personal and organizational environments, allowing users to seamlessly transition between independent projects and collaborative tasks.

The evaluation confirms the successful fulfillment of the outlined functional and non-functional requirements.

Zusammenfassung

Diese Arbeit untersucht die Integration von Multi-Tenancy-Funktionalität in Cloud-Anwendungen am Beispiel des PROCEED Management Systems (MS). Aktuell hat das PROCEED MS keine Unterstützung für Multi-Tenancy, wodurch kollaborative Arbeitsabläufe nicht möglich sind. Um diese Einschränkung zu beheben, wurden isolierte Umgebungen eingeführt, namens *Environments*, die es mehreren Nutzern oder Organisationen ermöglichen, ihre digitalen Güter innerhalb klar abgegrenzter, sicherer Arbeitsbereiche gemeinsam zu verwalten.

Eine hierarchische Ordnerstruktur wurde entwickelt, um hierarchische Strukturen von Organisationen abzubilden und das Management von digitale Güter zu verbessern. Das bestehende rollenbasierte Zugriffskontrollsystem innerhalb vom PROCEED MS wurde ebenfalls erweitert, um *Environments*- und ordnerspezifische Berechtigungen einzubeziehen.

Schwerpunkte der Umsetzung waren die Restrukturierung der digitale Güter- und Nutzermanagements sowie die Etablierung klarer und effizienter Datenbankschematas. Die entwickelte Lösung unterstützt sowohl persönliche also auch organisatorische Umgebungen und erlaubt es Nutzern, leicht zwischen unabhängigen Projekten und kollaborativen Aufgaben zu wechseln.

Die Evaluation bestätigt die erfolgreiche Umsetzung der definierten funktionalen und nichtfunktionalen Anforderungen.

Contents

1 Introduction

In today's digital age, businesses heavily rely on cloud applications to work on their assets, e.g. documents, spreadsheets, and presentations. Cloud applications are software tools that are accessed and run entirely over the internet. Typically, a copy of the application runs on a remote computer in a data center. Users access it through a web browser or a dedicated application. These tools represent a paradigm shift from traditional software applications, where the majority of the work was done on the user's device. Shifting a part, or even all the workload to a remote computer offers many advantages for both the users and the developers of applications:

- Accessibility: Cloud applications can be accessed anywhere from anywhere with an internet connection.
- Data safety: There is a big potential for increased data safety, as data can be stored in a secure data center, instead of on a user's device, which could be lost, stolen or damaged.
- Collaboration: Collaboration is easier for application developers to implement, as all users could interact with the same instance of the application, instead of having to manage communication between different local applications.
- Device-agnostic: Many cloud applications can be accessed from various devices because most of the functionality runs on a server and isn't dependent on the device, unlike traditional software, which often requires a large part of the application to be specifically programmed for each device.
- Low IT overhead: Users don't have to set up the application on their own, which would require technical knowledge. And in the case of cloud applications that are accessed through a browser, users don't have to install anything.

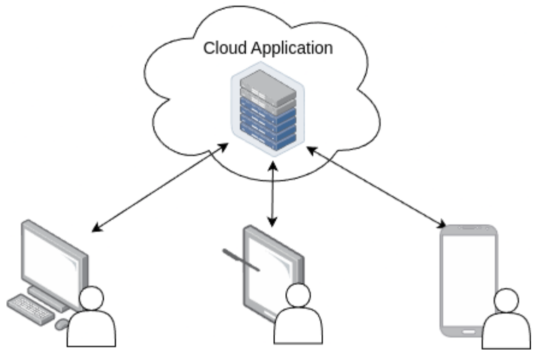

Figure 1.1: Users can access cloud applications from different geographical locations and from a variety of devices, such as laptops, smartphones, or tablets, as long as they have an internet connection.

One very common feature that makes these benefits possible is called *multi-tenancy*. *Multi-tenancy* is a software architecture in which an application can be used by multiple users or organizations at the same time. This has to be achieved without a new execution of the application, also referred to as an *instance*, for each user or organization. Without *multi-tenancy*, each user or organization would need to run the application on their own computers, negating most of the benefits listed earlier.

Think of an application that supports *multi-tenancy* like a big apartment building. Each tenant (user or organization) has their own private apartment (their space), where they store their belongings (their assets). Every tenant is living in the same building (cloud application), but each tenant has their own private space.

Many popular cloud applications use this approach. For example, when you use Microsoft Teams, Slack, or Asana, you're sharing the application with many other companies, but you only see and interact with your own team.

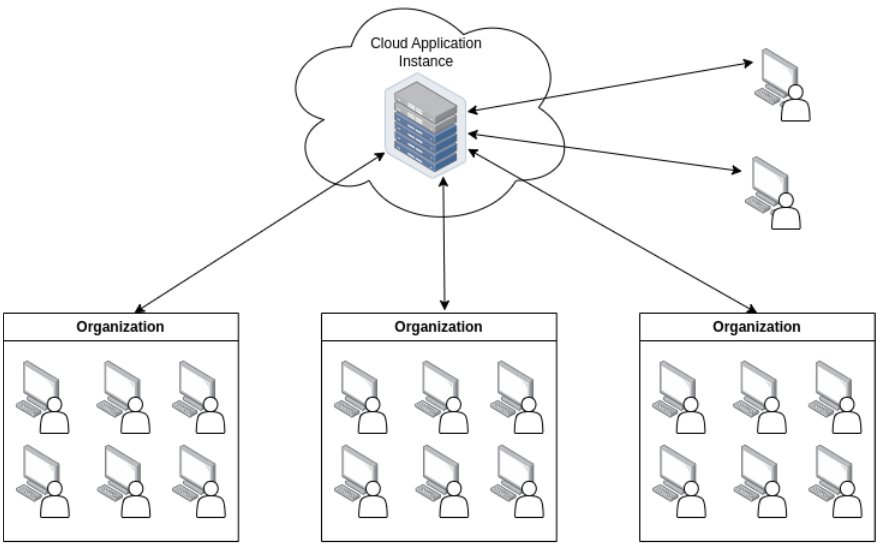

Figure 1.2: *Multi-tenancy* in cloud applications: the same instance of the cloud application, can be used by different tenants, with different structures, without them knowing about each other.

The PROCEED Management System (MS for short) is a cloud application for managing business processes. Business processes are sequences of tasks and decision points designed to achieve a specific business goal. PROCEED uses BPMN (Business Process Model and Notation), a standardized graphical notation, to represent these processes visually.

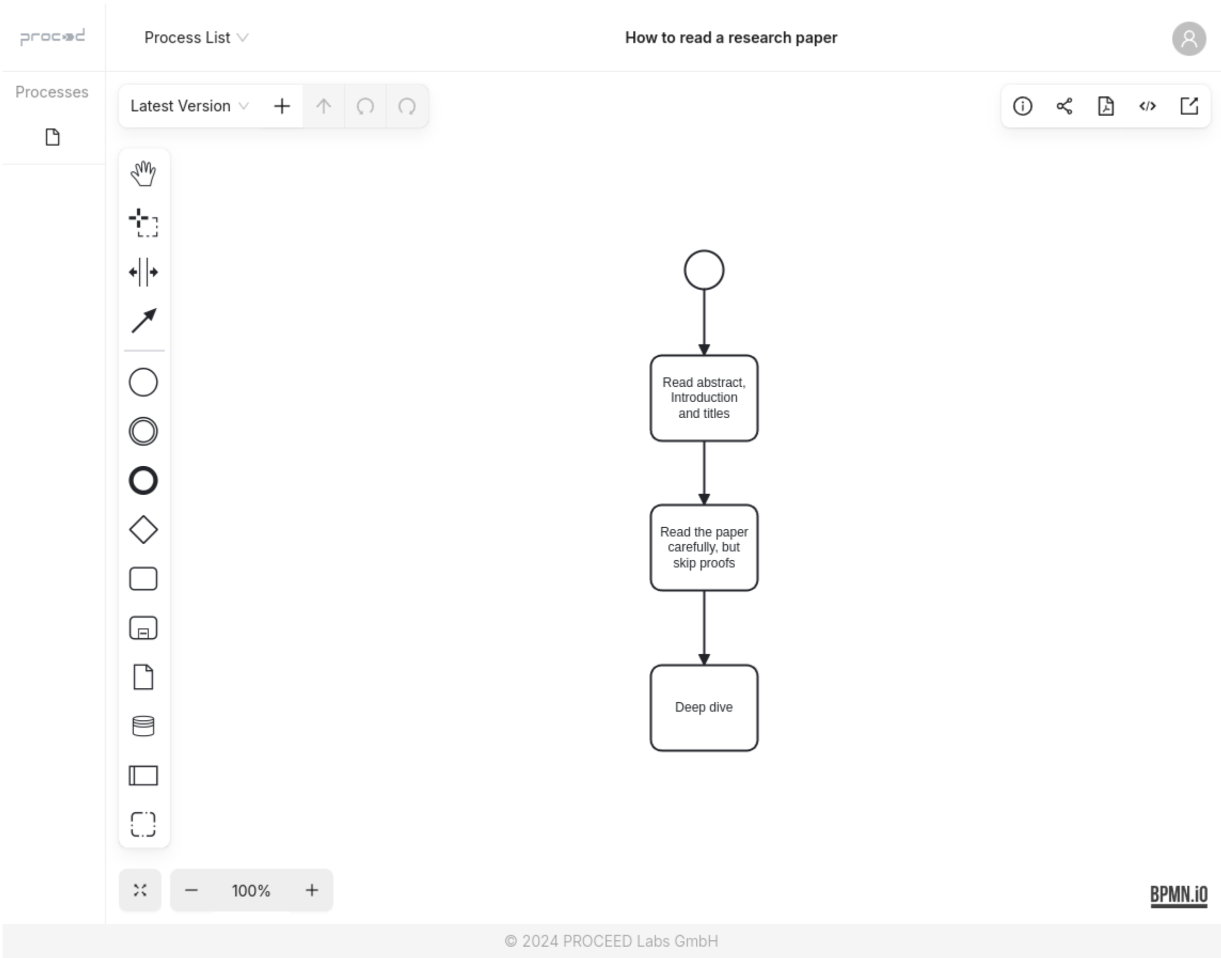

Figure 1.3: Example of process modeling in PROCEED.

Currently, the MS lacks full *multi-tenancy* support. It only supports individual users and doesn't fully support organizations. For organizations to be supported, members of the organization need to be able to have a shared workspace, where they can work on the same assets. However, the MS currently stores all assets in central place, and access to these assets is restricted through a permission system that allows users to see their own assets, or all assets if they possess the admin role. This means that users, regardless of if they are part of an organization or not, can either only see their own assets, or they can see all assets.

For this reason, this thesis will implement *multi-tenant* functionality into the PROCEED MS by introducing the concept of environments. Conceptually, environments are spaces where one or more users can work on shared assets. The MS will be able to hold multiple isolated environments. This way, an organization can have its own environment, where all members of the organization can work on shared assets.

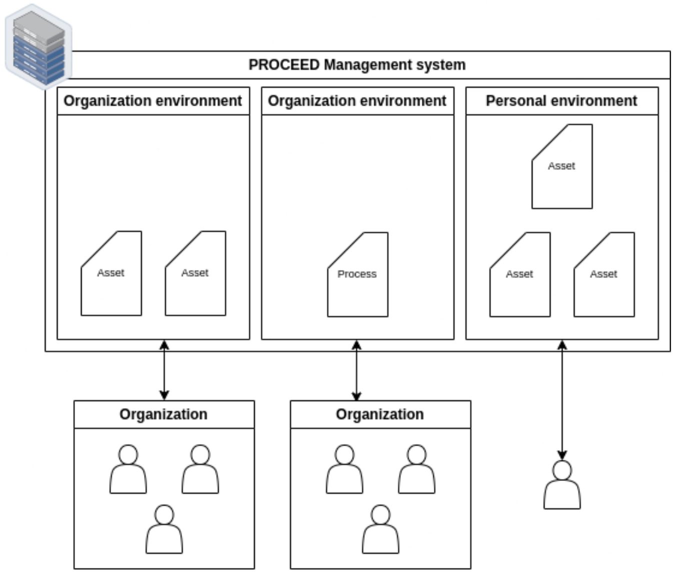

Figure 1.4: Goal of this thesis: tenants with different structures, can work on assets in their own isolated environments in the PROCEED Management System.

1.1 Research Questions

For environments to be successfully integrated into the PROCEED Management System, a few important questions need to be addressed. These questions will help ensure that environments work smoothly with the existing database structure, permission system, and user management. By answering them, we can ensure the implementation is effective

1. Environment representation: How can we integrate environments and their hierarchical folder structure within the MS' storage solution to ensure the following:
    - Data integrity: find a schema that facilitates data consistency after updates.
    - Asset-Environment association: find a schema that associates assets with their respective environments while maintaining a clear separation between multiple environments.
    - Efficiency: find a schema that allows to efficiently query the database.
2. How does PROCEED and its user storage have to be structured to allow users to be members of multiple environments?
3. How can organizations model their hierarchical structures within environments to allow members of the organization to have different levels of access to assets?
4. Users may want to create and manage their own projects independently of an organization: How can users work on personal projects outside an organization?
5. How can environments be implemented in a way that doesn't disrupt existing functionality and data structures?

1.2 Task List

The following task list outlines the concrete steps necessary to address the research questions and ensure a successful implementation of environments. Each task is designed to tackle specific aspects of the MS' architecture, user management, and role assignments.

1. The MS has to support environments, which should be isolated spaces where users can work on assets.
   a) Every asset in the MS must be stored in only one environment.
   b) Assets stored in one environment should only be accessed by members of that environment.
   c) Environments must have a hierarchical folder system to store assets.
      i. Find a suitable abstraction to represent folders in a database that facilitates consistency after updates and is fast to query.
      ii. Ensure privacy between environments: folders must only belong to one environment and should only store assets from that environment. Access to folders must be restricted to members of the environment.
2. The MS must be able to hold multiple environments and their assets.
3. The MS must allow different users to access different environments concurrently.
4. Implement personal and organization environments. While both are environments and share common functionality as described in 1, they must behave differently in some situations.
   a) Personal environments must only have one member.
   b) Personal environments should only store Processes and Folders.
   c) Organization environments should have a name and description.
   d) Organization environments support all assets described in 2.2.2.
   e) Organization environments must support multiple members
   f) Users must be able to create organization environments and invite new members.
   g) Organization environments must have a role system, where roles can be assigned to users, to manage their access to assets.
   h) Users of organization environments that have the right permissions must be able to invite users to the organization environment.
5. The MS' user management has to be adapted to fit environments: before the implementation of this thesis, users were strictly tied to one instance of the MS, meaning each time a new instance of the MS was created, a new user storage had to be configured.
   a) Users have to be global, meaning that they don't belong to any environment.
   b) Every user has to have a personal environment.
   c) Personal environments must be tightly coupled with users, i.e. when a user is deleted so is his personal environment.
   d) Implement Guest users.

i. The MS must support guest users, which are users that don't sign in with their personal information.
ii. The MS should offer a restricted set of functionalities to guest users.
iii. Guest users must have the ability to transfer their assets to a normal user.
iv. Guest users can must not be able to create or be part of organization environments.

6. The MS' preexisting role system must be adapted to fit organization environments and their folder structure: The MS already has a role system in place to manage users' access to resources, this has to be adapted to work with organization environments.
   a) Roles must belong to only one organization environment.
   b) The role system must be replicated for each environment, i.e. it works the same as before, with the difference that it is now specific to an environment. E.g. if a role allowed a user to manage all processes before the implementation of environments, now, with the same role, he will be able to modify all processes inside the organization environment the role belongs to.
   c) Find a suitable permission inheritance model for roles based on the folder structure of an environment (e.g. a user with a role in a parent folder, has the same permissions in all subfolders).
   d) Roles must always be enforced in the backend, to ensure privacy within organization environments.
   e) The frontend UI must adapt to a user's roles, by only showing him options that are allowed by his roles.

The following are non-functional tasks that have to be achieved with the implementation of environments in the MS. The goal of these is to ensure that the implementation is user-friendly and most importantly developer-friendly, as many developers will have to work with the codebase in the future. This means that where possible, simple solutions should be favored over complex ones.

1. Keep changes to the MS to a minimum.
2. The user interface for navigating and managing folders and environments should be intuitive and easy to use.
3. Prioritize developer experience by creating clear abstractions and APIs.
   a) The same data structure and functions should be used for both personal and organization environments where possible. E.g. the same function that creates a folder in a personal environment should be used to create a folder in an organization environment.
   b) Choose a simple data structure for the folder system, with straightforward functions for modification.
   c) Streamline environment identification and permissions check in the backend.
   d) Create simpler helper functions for the frontend of the MS, to adapt the interface to a user's permissions within an environment, than the existing functions for current

role system.

2 Foundations

To understand the further chapters, it is essential to first establish a foundational understanding of external technologies, and the core architectural elements of the MS. This chapter briefly introduces the OAuth 2.0 standard, then describes the MS' architecture and its role-based access control (RBAC) system.

2.1 JSON Web Tokens (JWT)

JWT is an open, industry-standard [RFC 7519] that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

A JWT consists of three parts separated by dots. Each part is a Base64Url encoded JSON string. The three parts are:

1. **Header**: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
2. **Payload**: The payload sections contains a JSON object, the contents of it are arbitrary.
3. **Signature**: The signature is used to verify that the payload of the JWT hasn't been changed. To create the signature, the header and the payload are concatenated with a period (.), this string is then either hashed with a secret or encrypted with a private key.

Once an application receives a JWT, it can compute the signature with the header and payload, if this matches the signature included in the JWT we can be sure that we issued the JWT and that the payload hasn't been tampered with.

2.2 OAuth 2.0 and OpenID Connect

OAuth 2.0 is an open standard for access delegation, commonly used as a way for users to grant client applications access to their information on other applications. OAuth 2.0 was born as a necessary security measure, to avoid sharing plaintext credentials between applications. Plaintext credential sharing, as outlined in [**?**] has many security risks:

1. Applications are forced to implement password authentication, to support the sharing of plaintext credentials.

2. Third party applications gain overly broad access to the user's account.
3. Users cannot revoke access to specific third party applications.
4. If any of the third party applications are compromised, the user's account is at risk.

OAuth 2.0 addresses these issues by decoupling the client application from the role of the resource owner, meaning that the client application will not get a full set of permissions to the user's account. Instead of users handing their credentials to third party applications, they can grant these applications limited access to their resources with an access token. This method avoids the user having to share his credentials with third party applications.

2.2.1 OAuth 2.0 Roles

OAuth 2.0 defines four roles for participants in the protocol flow:

1. Resource owner: The entity that can grant access to a protected resource, typically this would be an end user of a web application.
2. Resource server: The server hosting the protected resources.
3. Client: The application requesting access to the protected resources. OAuth 2.0 distinguishes between two types of clients: confidential and public clients. Confidential clients are capable of keeping their credentials confidential, while public clients, like browser-based applications, cannot.
4. Authorization server: The server that issues access tokens to the client after the resource owner has been successfully authenticated.

The resource server and the authorization server can be the same entity, but they are not required to be.

2.2.1.1 Authorization Code

Of the possible OAuth 2.0 authorization mechanisms, the Authorization Code grant is the most widely used for confidential clients (i.e., those capable of safely storing secrets). The Authorization steps are as follows:

1. **Client Redirects to Authorization Server**: The client first redirects the user to the Authorization Server's authorization endpoint. This request includes information such as the client's identifier, the requested scope of access, and the URL to which the Authorization Server should redirect once the user grants or denies permission.
2. **User Authenticates and Grants Access**: The user logs in (if they are not already) in the Authorization Server and decides whether to grant the client's request. If the user grants access, the Authorization Server returns only an authorization code to the client via the specified redirect URL.
3. **Client Exchanges Code for Access Token**: After receiving the authorization code, the client sends it to the Authorization Server's token endpoint, authenticating itself using its client credentials. In response, the Authorization Server returns an access token. The client then can use the access token to call APIs on the Resource Server.

2.2.2 OpenID Connect

OpenID (OIDC for short) Connect is an identity layer built on top of the OAuth 2.0. OIDC uses as its base the Authorization Code flow, and it introduces a new type of token called an ID Token. This ID Token is a JSON Web Token (JWT) that contains minimal information about the authenticated user. Most importantly, the ID Token carries a Subject ID, which is a unique identifier for the user in the Identity Provider's system. The ID Token is sent alongside the Access token to the client. Through the Subject ID the client can authenticate the user, since it came from a trusted source, and sign them in to their platform.

2.3 MS Architecture

The MS is built using Next.js [1], a React [2] framework that allows for server-side rendering. Although Next.js' architecture is different from traditional server-side rendered applications and single-page applications, for the purposes of this thesis, it can be thought of as being split into a single-page frontend and a backend. The frontend executes JavaScript code in the user's browser, and is responsible for rendering the UI, handling user input and making requests to the backend. The backend runs on a server and is responsible for handling requests from the frontend, e.g. saving or querying data.

2.3.1 PROCEED MS' Storage Solution

The PROCEED MS doesn't use a database management system (DBMS) like MySQL or PostgreSQL to store its data. Instead, it stores its data in multiple JSON [3] files. While these files allow for a very flexible data structure, the MS stores one array of objects per file. The objects found in each file have the same structure, making each file comparable to a table in a relational database. Each object will be called an entry from now on. The MS uses Zod [4] to enforce that the data that is being stored follows a specific structure. Zod is a schema declaration and validation library, it allows the MS to define the shape of JSON serializable data. For purposes of simplicity, when we talk about a schema, instead of showing the code that describes the Schema, we will show the typescript type that is inferred from the schema.

```
import { z } from 'zod';

const UserSchema = z.object({
  id: z.string(),
  username: z.string(),
  image: z.string().optional(),
})

// TypeScript type that satisfies the UserSchema
type User = {
    id: string;
    username: string;
```

[1] https://nextjs.org/
[2] https://reactjs.org/
[3] https://datatracker.ietf.org/doc/html/rfc8259
[4] https://zod.dev/

```
    image?: string | undefined;
}

```

Listing 2.1: Example of a Zod schema and the corresponding TypeScript type.

2.3.2 PROCEED's Assets

Assets are objects that users can create and manage through the MS' interface. They are the core "product" that PROCEED offers, i.e. the focal point of its value proposition, unlike management assets, which regulate how a user can use the MS. When referring to assets, we are only talking about the core features of the MS, not objects that aid in the usage of the MS, like Roles .e.g., which only help with managing access to assets. Currently, the MS supports the following assets:

1. Processes, Project and Templates: These assets store BPNN at their core.
2. Machine: Asset that represents a server running Distributed Process Engine.
3. Execution: An execution represents a process that is being executed distributedly.

Furthermore, the MS implements management assets, which are used to control how users interact with the MS. These are considered assets because users can directly manage them. Currently, the MS includes the following management assets:

- Role 2.3: roles are used to manage how users can access assets.
- RoleMapping: role mappings are used to assign roles to users.
- User: represents a user's personal information, e.g. name, username and email.

2.4 PROCEED's Role System

The PROCEED MS uses a Role-Based Access Control system to manage user authorization and determine what actions a user can perform. Roles can be seen as bundles of permissions, which are granted to users. A user can have multiple roles and all the permissions of the roles are additively combined. That is, by adding a permission, a user can never do less than before. Typically, roles are assigned to users based on their job function. RBAC can be advantageous since roles can be assigned to multiple users and don't change often, making them easier to manage than individual permissions.

2.4.1 MS' Role System Terminology

The following terms are important to understand the role system in the MS:

- Resource: A resource is any protected entity in the management system, that can be accessed by users. Resources can be either assets or management assets.
- Action: An action is a specific operation that can be performed on a resource, e.g. view, update, create, delete.

- Permission: A permission is a tuple consisting of resource type and a list of actions, which specifies that a user can perform the actions on the resource instances. Optionally a permission can have conditions that have to be met the by resource instances, for the user to be able to perform the actions.
- Role: A role is a set of permissions. Roles can be assigned to users, which then inherit the role's permissions. Roles can have expiration dates, after which all permissions are revoked.

2.4.2 MS' Resources and Actions

The following are the resource types that are used in the PROCEED MS: `Process`, `Project`, `Template`, `Machine`, `Execution`, `Role`, `User`, `RoleMapping`.

These are the actions that can be performed on these resources: `none`, `view`, `update`, `create`, `delete`.

2.4.3 Role Mappings

RoleMappings are a management asset, that is used to assign roles to users. RoleMappings store a user identifier ID and a role ID.

2.4.4 MS' Roles in CASL

The PROCEED MS uses CASL [5] to implement Roles. CASL is an isomorphic authorization JavaScript library. To enforce authorization CASL has *abilities*, which are assigned to users. Abilities expose functions to check weather a user can perform an action on a resource. Abilities are made up of rules, which are defined by four parameters: user action, subject, conditions. User actions and subjects are analogous to actions and resources 2.3.1.

CASL differentiates between subject type and subject instance. A subject instance is a specific instance of a subject type, e.g. a specific process users are working on, is an instance of the resource type "Process".

Conditions are used to specify additional conditions that have to be met by a resource instance, for a user to be able to perform an action on it. E.g. a user can only update a process if he created it.

```
import { defineAbility } from '@casl/ability';

class User {
  constructor(id) {
    this.id = id;
  }
}

class Process {
  constructor(user, name) {
```

[5] https://casl.js.org/v6/en/

SNET
SERVICE-CENTRIC NETWORKING

```
    this.authorId = user.id;
    this.createdOn = new Date();
    this.name = name
  }
}

function abilityForUser(user){
  return defineAbility((can, cannot) => {
    can('delete', 'User', {id: user.id});

    can('update', 'Process', ['name'], {authorId: user.id});
  });
}

const user1 = new User(1);
const user1Ability = abilityForUser(user1);
const user1Process = new Process(user1, 'some process');

const user2 = new User(2);
const user2Ability = abilityForUser(user2);

user1Ability.can('update', 'Process'); // true
user1Ability.can('update', user1Process, 'name'); // true
user1Ability.can('update', user1Process, 'createdOn'); // false

user1Ability.can('delete', user1); // true
user1Ability.can('delete', user2); // false

user2Ability.can('update', 'Process'); // true
user2Ability.can('update', user1Process); //false
```

Listing 2.2: CASL example

If there exist any possible resource instance, for which a user has permission to perform an action, then the user has permission to perform the action on the resource type. E.g if a user has permission to view some process in the MS, then he has permission to view the resource type "Process".

3 Concept and Design

This chapter outlines the key components of the implementation of environments in the MS. The core components are users, environments, roles, and assets. In essence the concept can be summarized as follows: users can be part of multiple environments, which hold Assets. Users that are part of an environment can work on the assets that are stored in it. Each environment has a set of Roles that determine what their users can do with its assets. All other components that will be introduced will help to manage and enforce these relationships. Figure 3.1 shows the relation between users and accounts.

3.1 Conceptual Overview

Environments will be either personal or organization environments. Personal environments have an explicit owner and organization environments can have multiple members which are associated to the environment via a new management asset called membership. Every environment will have an ID. All assets and management assets will store an environment ID, which will link them to only one environment, this includes roles and role mappings. Folders will be added to the MS' assets and thus will also store an environment ID. Each folder stores a reference to its parent folder. Processes will store a folder ID under `folderId`, which will be used to determine the folder they belong to. Figure 3.1 shows these relationships.

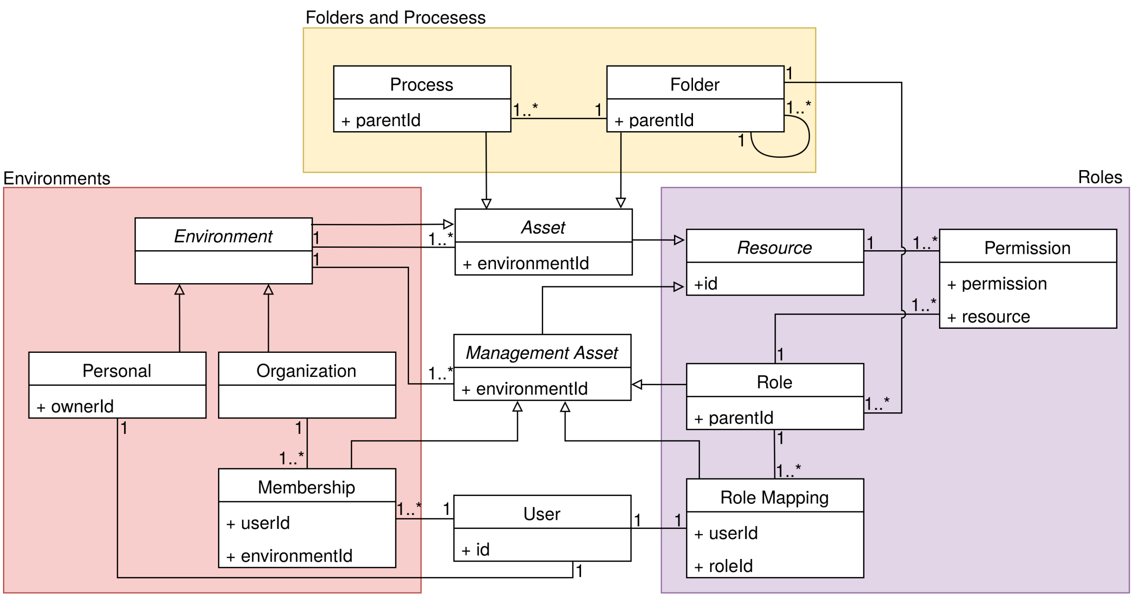

Figure 3.1: Relation between users, environments, processes, folders and roles.

3.2 Modifications to Assets and Resources

This thesis will modify all assets and management assets 2.2.2 that are supported by the MS, these will be modified to be contained inside environments. Additionally, environments will be added to the MS' assets and resources. As will be explained in 3.4, Users won't belong to a single environment, they can instead be members of multiple environments, for this reason, users will be removed from the MS' assets and resources. Furthermore, folders and memberships will be added to the MS' assets and resources.

3.3 Data Storage Approaches for Isolated Environment Data

As outlined in [3, 3.3], data leaks between tenants are very dangerous. For this reason isolation between tenants is crucial. The first step to ensure isolation is to choose a good data storage model. There are mainly three models to choose from as described in [4, 4.1]:

- **Separate databases**: Each tenant has a dedicated database instance, offering complete physical isolation. This model offers a high isolation level, however it is resource-intensive and it involves a high operational overhead.
- **Shared database, separate schemas**: All tenants share the same database, but each has a distinct schema. This approach balances isolation and resource efficiency, with less complexity than fully separate databases.
- **Shared database, shared schema**: All tenants use the same database and schema, with tenant data identified by a tenant ID. This model maximizes resource efficiency but requires robust access control and query isolation mechanisms to ensure tenant boundaries are respected.

Since the first two models are resource-intensive and introduce operational overhead, we chose the third model: **Shared database, shared schema**. The trade-off is a lower level of data isolation compared to the other two approaches. To mitigate this, every asset in the system explicitly stores an `environmentId`, which is always validated during data access and modification operations. This strict enforcement ensures tenant boundaries are always respected.

3.4 New User Management System

Previously all users in the MS were members of a single organization. In order for users to be part of multiple organizations, they can't be tied to a single organization. As a part of the implementation of environments, users are now independent of organizations, they represent individual people utilizing PROCEED and are stored as entries in the MS' storage solution.

To facilitate the exploration of the MS, without creating a user, we introduced the option to use the MS as a guest. Thus, we differentiate between authenticated users and guest users. Guest users can only use a subset of the MS' features. Guest users can transition to being authenticated users whilst retaining their assets, to do this they will need to sign in with their personal data.

All users have a personal environment 3.6.1 in which they can create and manage assets freely. Authenticated users can also be part of and create organization environments 3.6.2, where they can collaborate with other users.

3.4.1 Authenticated Users and Accounts

To allow the same user to be able to sign in with different Oauth 2.0 providers, e.g. with Google or Facebook, we store a separate record, called *account*, for each of the user's sign-in methods. This means, that the relationship between users and accounts is one-to-many, a user can have multiple accounts, but an account can only be linked to one user. When a user signs in with a Oauth 2.0 provider, the provider returns a user ID, then the MS searches for an account with the same user ID. From that account the MS can find the user.

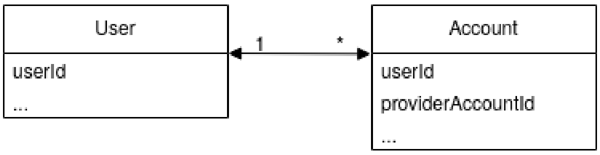

Figure 3.2: Relation between users and accounts.

3.4.2 Merging a Guest User with an Authenticated User

As previously stated, a guest user can transition to being an authenticated user by signing in with his personal data. Afterward the user will be marked as authenticated and will be able to use the MS freely. However, it could be the case that his personal data already corresponds to an existing user. In this case, the user will be asked if he wants to merge his assets with the existing authenticated user. If he chooses to do so, all the assets in the guest user's personal environment will be transferred to the authenticated user's personal environment. Otherwise, all the assets created by the guest user will be deleted.

3.4.3 Guest User Storage

For storing guest user data, one could take one of two approaches: storing the data in the user's browser or storing it in the MS' database, alongside the data of authenticated users. Storing the data locally has two benefits: The MS doesn't have to store data of users who might never return and the MS would become less susceptible to an attack where the attacker tries to use up as much space as possible in the MS' storage solution. However, this approach has one key downside, the MS would have to implement two storage solutions and the frontend would need to switch accordingly between them. The added complexity would make it harder for developers to get an overview of the MS' codebase and it would mean that any future changes will have to be implemented twice. For this reason, we determined that storing guest user's data locally isn't worth the benefits. We decided to store guest users data in the MS the same way we do it for authenticated users, with the difference that a flag is set in the user entry to indicate that they are a guest. This way, all the endpoints that authenticated users can call to interact with the MS can also be called by guest users. An important caveat is that, to enforce some of the feature restrictions, relevant endpoints have to check whether the user is a guest.

Furthermore, the MS needs to clean up inactive guest users, to prevent the MS' storage solution from filling up with unused data.

3.4.4 Development Users

During development, it is often inconvenient or impractical for developers to configure all the necessary environment variables required for full authentication functionality. For instance, integrating OAuth 2.0 authentication (see Section **??**) or enabling email-based sign-in requires valid credentials, secrets, and sometimes external services that may not be accessible in a local development environment.

To streamline testing and development without depending on such services, the Management System (MS) implements two predefined development users: *johndoe* and *admin*. These users are only available when the MS is running in a development environment.

3.5 Hierarchical Folder Structure

Folders will be added to the MS' assets, they are intended to allow organizations to mirror their hierarchical structure and to facilitate the organization of assets in general. Starting from a root folder, users are able to store assets within folders and nest folders inside other folders, creating a flexible and intuitive structure. In this thesis, folders were only implemented to support processes, but they could be extended to support more of the assets that were described in 2.2.2.

3.5.1 Folder Structure Storage Model

The folder structure is essentially just a rooted tree. The MS doesn't use a database management system, thus it doesn't use a relational language like SQL, still, the data is stored in a way that mimics a relational model. For this reason the rooted folder tree needs to be mapped to a relational model. As outlined by Joe Celko in [5, 28], there are mainly three ways to model trees in relational model: Adjacency List, Nested Set, and Path Enumeration, more commonly known as Materialized Path. Each of these models stores a node as a single entry, but they differ in how they encode their position in the tree:

- **Adjacency List model**: Each node stores an identifier and a reference to its parent.
  - Advantages
    - Finding a node's children is trivial.
    - Finding a nodes' parent is trivial.
    - Adding a node to the tree is trivial and doesn't require updating other nodes.
    - Moving nodes and their subtrees is trivial, since it only requires updating the parent reference. However, a check has to be made to ensure that the node's new parent isn't one of its descendants, thus creating a cycle.
  - Disadvantages
    - Finding a node's descendants requires a recursive query, however, this is bounded by the depth of the tree and the amount of nodes.

    * Finding a node's ancestors, i.e. all the nodes in the path from the root to the node, requires a recursive query, however, this is bounded by the height of the tree and should be fairly efficient.
    * Removing a node requires a recursive query to also delete its descendants, however, this is bounded by the depth of the tree and the amount of nodes.

- **Nested Set model**: Each node has a left and right value, describing an interval starting from the left value and ending at the right value. Children nodes' intervals are contained within the parent's interval. Furthermore, the intervals of siblings are disjoint.
  - Advantages
    * Finding the descendants of a node is trivial, the query has to select all nodes, whose interval is contained within the node's interval.
    * Finding a node's ancestors is trivial, the query has to select the all the nodes with a left boundary smaller than the node's left boundary.
    * Finding a node's parent is trivial, the query has to select the node with the smallest left boundary that is smaller than the node's left boundary.
    * Deleting a node and its subtree is trivial, the query is analogous to finding the descendants of a node.
  - Disadvantages
    * Finding a node's children requires a complex query, since it has to select only the node's children without also including the descendants of the children. This query can be inefficient because it has to apply additional filtering to retrieve only the direct children. This adds complexity and can slow down performance.
    * Adding nodes is non-trivial, since it requires knowledge of its siblings to avoid overlapping intervals. Additionally depending on the implementation, it might require updating intervals of other nodes.
    * Moving nodes and their subtrees is non-trivial and inefficient, since it requires updating the intervals of the moved node and all of its descendants. Depending on the implementation it may even be required to update the intervals of other nodes.
- **Materialized Path model**: Each node stores the path from the root to itself.
  - Advantages
    * Finding a node's ancestors or its parent is trivial, since the path is explicitly stored, the ancestors of a node can be retrieved simply by parsing the stored path.
    * Finding a node's descendants is trivial, they can be found by querying for nodes whose paths start with the current node's path. However, depending on the implementation, checking each node's path can be inefficient.
    * Adding nodes is trivial, as the node's path can be constructed by appending the node's identifier to the parent's path.
    * Removing a node and its subtree is trivial, as the node's descendants can be easily queried.
  - Disadvantages

* Moving nodes and their subtrees is inefficient, since it requires updating the path of every node in the subtree.

* Each node stores a string with its path, which can be inefficient in terms of storage space, especially for trees with a large depth.

For an example of the three models, see Figure **??**. A simple comparison of these models is presented in Figure 3.4.

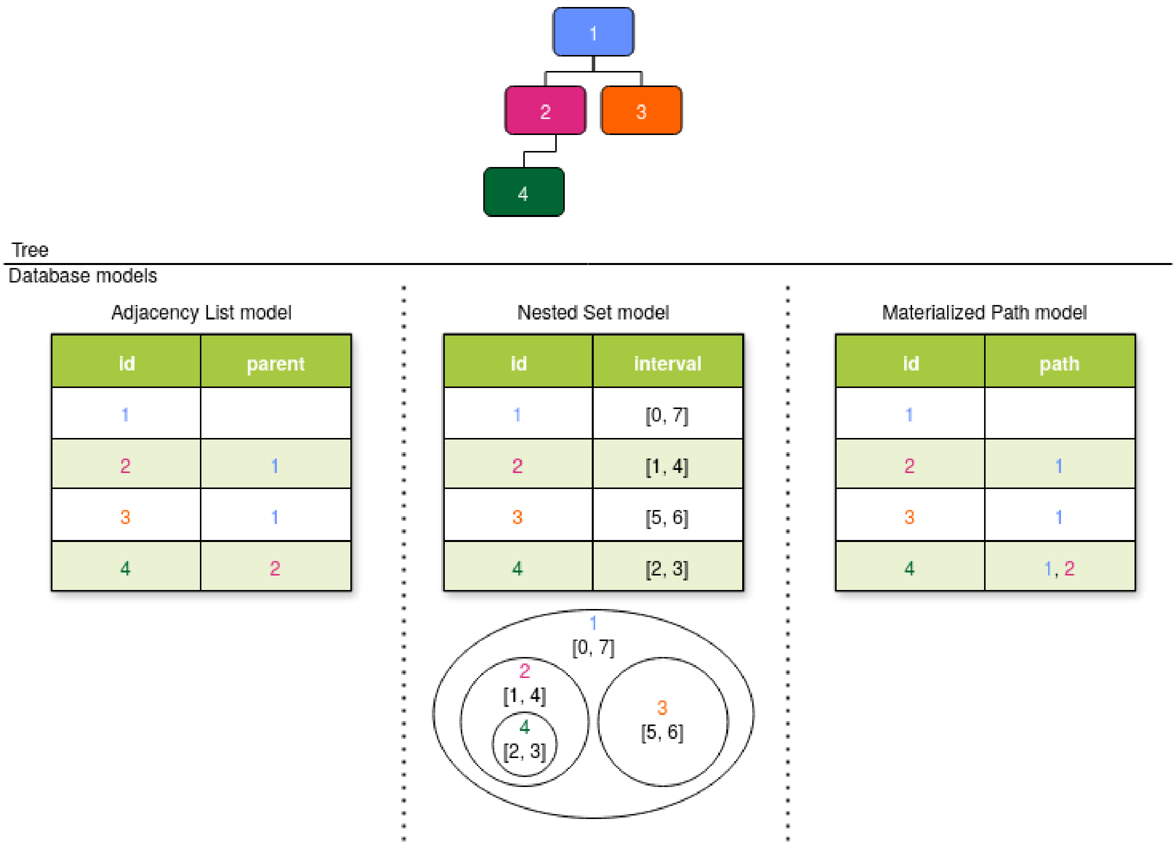

Figure 3.3: Comparison of the Adjacency List, Nested Set, and Materialized Path models.

	Adjacency List	Nested Set	Materialized Path
find children	✓ ✓	X X	✓ ◯
find descendants	✓ ◯	✓ ✓	✓ ✓
find parent	✓ ✓	X X	✓ ✓
find ancestors	✓ ✓	✓ ✓	✓ ✓
add nodes	✓ ✓	X ◯	✓ ✓
remove nodes	✓ ◯	✓ ◯	✓ ✓
move node	✓ ✓	X X	◯ X

✓ _: Easy query ◯ _: Moderately complex query X _: Complex query
_ ✓: Efficient _ ◯: Moderately efficient _ X: Inefficient

Figure 3.4: Comparison between Adjacency List, Nested Set and Materialized Path models.

To choose the right model, we have to consider the requirements of the MS. Users need to be able to view, add, delete and remove folders. It becomes immediately clear, that the Nested Set model is not suitable for the MS, as adding, removing and moving nodes is inefficient, this model is more suited for read-heavy workloads. That leaves us with the adjacency list and materialized path models. The materialized path model has two small advantages: finding descendants and removing a folder together with its subtree is easier. Both queries only require simple string comparisons. However, the adjacency list model isn't far behind on those two points, and is substantially more efficient when it comes to moving nodes. Furthermore, the adjacency list model is better at finding children, which is more valuable to the MS than finding descendants, as the Process view will only show the children. Additionally, the adjacency list model is more space-efficient, as it doesn't store its path, this can be important for trees with a large depth. For these reasons, the MS will use the adjacency list model to store the folder structure.

3.5.2 Storing Assets Inside Folders

After the folder structure is implemented, it is necessary to store assets inside folders. This thesis only implemented this feature for processes, however, the same principle could be applied for other assets. A complete redesign of the process' data structures isn't feasible, as it would require rewriting a large part of the MS. For this reason a simple expansion to the data structure was chosen, were analogous to the adjacency list model, each process stores a reference to the folder it is stored in. This approach allows a folder to be moved without requiring updates to its descendants. Additionally, moving an asset to another folder only requires updating the asset's reference to its folder.

3.6 Environments

Conceptually, environments are where everything except users are stored. Users aren't stored in environments as they can be a part of multiple environments. Instead, the MS stores memberships, which specify that a user is part of an environment.

There are two types of environments, personal and organization environments. Personal environments are intended for personal use and organization environments are intended for organizations.

3.6.1 Personal Environments

Personal environments are assigned to each user once they sign in. The user for which the environment is created is the only member of this environment, and is therefore called the owner. No other users can be a part of this environment. Personal environment only allow users to create and manage processes and folders, while other features offered by the MS can only be used in organization environments 3.6.2.

3.6.2 Organization Environments

Organization environments are intended to be used by organizations, thus they can have a name, description and a logo. In contrast to personal environments, users are allowed to use all the MS' features in an organization environment.

Organization environments can also have multiple Users that are part of it, these are called members.

3.6.3 Environment Memberships

To keep track of which users are part of an environment, a new management asset will be added: memberships. Each membership links one user to one environment, specifying that the user is part of that environment.

3.6.4 Storing Assets inside Environments

As previously stated, all assets within the MS 2.2.2, including folders, will be modified, so that each asset instance establishes a clear association with a single environment. Every asset will store a reference to the environment it belongs to. This reference is immutable, with the only exception being when assets are transferred from a guest user to an authenticated user 3.4.2.

Processes and folders will be contained within folders, which implies that they belong to the same environment as their root folder. This means that for these assets, we are storing the environment they belong too twice. Storing redundant information is risky as it can lead to inconsistencies if updates aren't done correctly. For instance if a folder's environment reference is changed, but those of its children are not, then the folder structure would span across two environments. However, this is not a practical issue in our implementation for two reasons. First, an asset's environment reference is never changed after creation, so there's no risk of an update causing an in inconsistent state. Second, the only exception to this, is when the reference is modified is during the transition from a guest user to an authenticated one, which involves transferring all the user's assets. Outside this specific case, the remains constant, ensuring structural consistency.

3.6.5 Environment Selection

Users can be members of multiple environments and must be able to work within each of them. There are two ways of accomplishing this: users can select one environment at a time, or they can work on multiple environments at the same time. Environments contain many features and views, making it unfeasible to show them all simultaneously for many environments. Thus, some level of selection is necessary to avoid cluttering the interface.

This selection could be granular, where the user selects per view which environment he wants to work on, however this could lead to confusion, if the user switches views and forgets that he's working on a different environment. For this reason, we chose to have a global environment selection, i.e. all the elements in the UI will show the assets and views of one environment.

The environment selection can be either implicit or explicit. In the implicit method, the selected environment is not reflected to the user in the URL, this could be accomplished by storing the selected environment in the user's cookies or in the browser's local storage for example. In the explicit method, the environment is encoded in the URL, making the current environment clearly visible. Of course a combination of both methods is also possible, but in order to keep the implementation simple, we will only choose one. The implicit method has the advantage that the URLs are shorter and easier to read, however it has the disadvantage, that some URLs can't be shared, since the implicitly selected environment of another user might not be the same. For this reason, the explicit method was chosen as it allows users to share links with the cost of longer URLs and because it's more transparent to the user.

3.7 Role Based Access Control within Environments

Roles define what actions a user can perform on assets. Prior to the changes introduced in this thesis, roles in the MS 2.3 were global, meaning their permissions applied universally to all assets across the system. However, with the addition of environments and a folder structure, this approach is no longer practical. Roles will now be tied to a specific organization environment, restricting their permissions to assets within that particular environment. In personal environments, where there is only one user, the user will have full control and be able to perform all actions on his assets without restrictions, this is done internally by giving the user a role that has all permissions.

Folders allow organizations to mirror their hierarchical structure, but this wouldn't be entirely useful if roles were applied to all assets inside the environment. For this reason, roles can now be associated to a folder. A role can define permissions for many assets, of which not all can be stored in folders. If a role is associated with a folder, then, only the permissions that are for assets that can be stored in folders, will be affected by this association. The permissions of roles that are associated with a folder cascade down the folder structure, i.e. the permissions of a role associated to a folder will apply to all the descendants of that folder. In this thesis, the folder structure was only implemented for processes, this means that only the permissions for processes and folders will cascade down the folder structure. Roles that aren't associated to a folder will be applied to all assets in the environment.

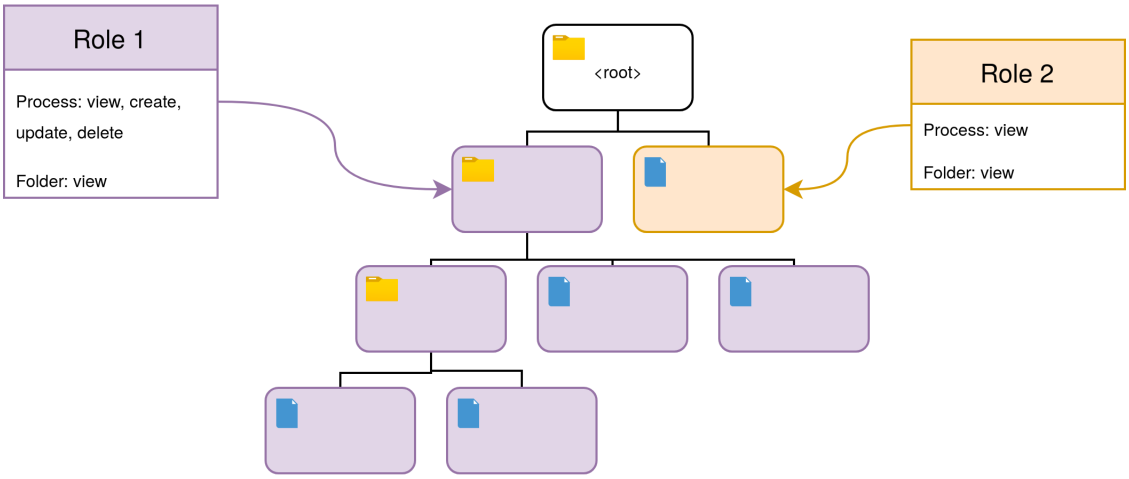

Figure 3.5: Permissions of roles cascade down the folder structure.

If a user's role allows him to view assets and is associated to a folder, then the user also has the permission to view all ancestors of the folder. But this permission is only restricted to the parent folders themselves, not their contents. This allows users to navigate the folder structure until they reach the assets they're allowed to view and manage.

3.7.1 New Resources

With the introduction of environments, all the previously available resources still hold the same meaning, with the difference that they are now associated with an environment. Additionally, two new Resources were introduced, for which roles can specify permissions:

- **Folder**: Folders are used to organize assets in an environment. If a role is associated to a folder, then the permissions for the folder apply to that folder and all folders
  - `view`: Allows the user to view the folder, its name and description. This doesn't necessarily mean that the user can view the assets inside the folder, as he still needs the `view` permission for each child.
  - `create`: The user can create new folders inside the folder he has this permission for.
  - `update`: The user can update the folder's name and description, as well as move the folder, to another folder where he has the `create` permission.
  - `delete`: The user can delete the folder.
- **Environment**: Environments are isolated workspaces inside the MS.
  - `view`, `create`: These actions are not implemented for environments. Each member can view the name and description of the environment. `create` can't exist as it

implies that the environment doesn't exist yet, and therefore a role that contains it cannot exist.

  - `update`: The user can update the environment's name, description and contact number.
  - `delete`: Allows the user to delete the environment with all its assets, this is only possible for organization environments.

3.7.2 Enforcing Permissions Based on the Folder Structure

In order to enforce permissions based on the folder structure, it is necessary to fetch the newest state of the folder structure every time a user wants to perform an action on an asset. Roles can't store a representation of the folder structure, as it might become outdated. Since permissions need to be verified, for many requests in the MS, and also in the user's browser, to adapt the UI, we decided to compute and cache a representation of the folder structure of every organization environment, from which an asset is requested. This representation is also sent to the user's browser, for the UI to be able to adapt based on the user's permissions.

3.7.3 Default Roles

For each organization environment two roles will be created, which cannot be deleted and cannot be associated to a folder:

- `@admin`: This role has all permissions for all assets in the organization environment and it is first assigned to the user that creates the organization environment. Only users with the `@admin` role can add new users to this role.
- `@everyone`: The permissions in this role apply to for all the users that are part of the organization environment. The permissions in this role start out empty, but can be modified.

4 Implementation

This chapter describes how the concepts outlined previously were translated into practical solutions within the MS. It details the key technical details involved in integrating environments, folder structures, user management, and the enhancements made to the role-based access control system. The implementation ensures robust isolation between environments and aims to achieve high maintainability and simplicity to support ongoing development.

4.1 Users

The user management in the MS was completely reworked to support the new environments design structure. Users are now stored in a separate file in the MS' storage solution and are independent of any environment. Additionally, guest users were added to the MS to allow users to try the MS without signing in with their personal information.

4.1.1 Sign In Flows

User authentication is implemented by leveraging OpenID Connect 2.1.2, with the help of the NextAuth.js [1] library. A JWT token [2] is stored in the user's browser cookies [3], which is then parsed and verified by the MS' backend. This token contains the user's ID. If a user makes a request to the backend, with a valid JWT token the user is considered authenticated. If a user couldn't be authenticated, they're redirected to the sign-in page.

NextAuth.js has many sign-in methods built-in, which can be set up with little configuration. For email-sign in, NextAuth.js sends an email with a link to the MS' backend with a token. When the backend receives this token, the user's cookies are set, and they're authenticated.

NextAuth.js provides hooks that allow us to customize the sign-in flow, the most important one for this thesis is the `signIn` hook. This hook is called when a user tries to sign in, this can be a new user or a returning one. As arguments, the `signIn` hook receives the user's data, and the account that the user is trying to sign in with, if the user doesn't exist the user data may be empty. If the hook returns true, the sign-in flow continues, if it returns a string or false, then the sign-in flow is stopped, and the user is redirected to an error page.

[1] `https://next-auth.js.org/`
[2] `https://www.rfc-editor.org/rfc/rfc7519.txt`
[3] `https://www.rfc-editor.org/rfc/rfc7519.html`

4.1.2 Authenticated Users

Authenticated Users are users that sign in to the MS either with their email or with a OAuth 2.0 provider. They're stored in the MS after they have successfully signed in. For authenticated users we store an ID, a flag named `isGuest`, set to false, and their personal information. This is the schema for authenticated users:

```
{
    id: string;
    isGuest: false;
    firstName?: string | undefined;
    lastName?: string | undefined;
    username?: string | undefined;
    image?: string |  undefined;
    email?: string | undefined;
}
```

Listing 4.1: Schema for authenticated users.

All the personal information is optional, because depending on how the user signs in, the information might not be available, when the user is created. After the user is created, if he has missing information, we prompt him to fill it in.

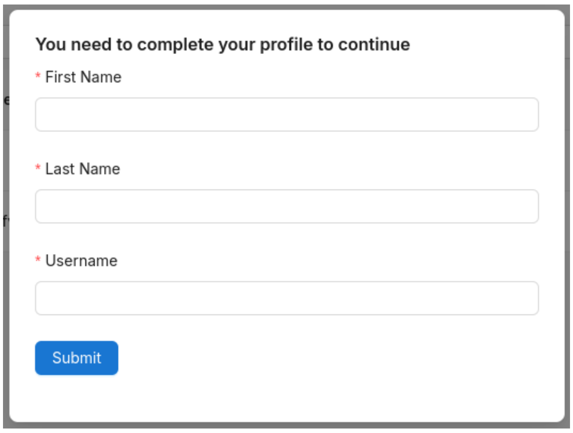

Figure 4.1: Prompt to fill in personal information.

The `image` field is used to store a URL to a user's profile picture. This field can only be set if the user signed in with an OAuth 2.0 provider, that provided a profile picture URL. We don't accept custom URLs, as this allows for attackers to supply URLs to a server they control, which can be used to track other user's browsers. By only saving URLs provided by trusted OAuth 2.0 providers, we ensure that the source of the image is reliable.

As stated before, Accounts are used to link sign-in data provided by OAuth 2.0 providers after someone signs in, to a user in the MS. OAuth 2.0 providers provide a unique ID for the user that signed in, to recognize a user's account we need to store this ID and the name of the provider. Additionally, to link the account to a user, we store the user's ID in the MS. The schema for accounts is as follows:

```
{
```

```
    id: string;
    type: "OAuth";
    userId: string;
    provider: string;
    providerAccountId: string;
}
```

Listing 4.2: Schema for accounts.

4.1.3 Guest Users

Users that aren't signed in can choose to use the MS as a guest, this doesn't require the user to input any personal information. For these users, a new user is created and stored in the MS. These users only store a unique ID and a flag named `isGuest`, set to true.

```
{
    isGuest: true;
    id: string;
}
```

Listing 4.3: Schema for guest users.

After a user signs in as a guest, a JWT token pointing to a real user is stored in his cookies, so he can use the MS like an authenticated user. Since there is no personal information stored, there is no way for the user to sign in to his guest user from another device. This also means, that if the JWT cookie is deleted, the user will lose access to his guest user.

4.1.4 Enforcement of limits for guest users

Guests can have their own personal space for their assets, but they aren't allowed to create or join organization environments. The system specifically prevents these actions in the corresponding endpoints, by checking the `isGuest` flag. Consequently, any features inside an organization environment are inaccessible to guest users.

4.1.5 Merging Guest User's Data with an Authenticated User's Data

As explained in 3.4.3, a guest user can sign in with an email or an OAuth 2.0 provider. Once signed in, the `isGuest` flag is set to false, and the user's sign-in method is stored. Since the user ID doesn't change, there's no need to transfer existing assets. If the chosen account is already linked to another user in the MS, then the user is redirected to a page where he can choose to either transfer the guest user's assets to his already existing account or discard them. This page takes a JWT token as a query parameter, which is generated after the user signs in. In the case of an Email sign-in, the token is generated by the MS when the user requests the sign-in link, and added to the sign-in link as a URL that will be called after authenticating the user.

we store a `signedInWithUserId` value to reference that user. This reference is needed, for the case that a user tries to sign in with an email, a This flag is needed to prevent the attack shown in ????? where a user could hijack a guest account's data through manipulating the

Afterward, the user can decide whether to transfer the guest's assets to their personal environment or discard them.

Guest users can choose to sign in with their email or with an OAuth 2.0 provider. By doing so, their guest user is turned into an authenticated user. The `isGuest` flag is set to false, and their sign-in method is stored. All their assets don't need to be transferred, as they are already stored in the MS, and the user ID wasn't changed. This approach only works if the account that the user used to sign in, wasn't already linked to an authenticated user. If the account is already linked to an authenticated user, we store a new value in the guest user's entry, named `signedInWithUserId`, that references the authenticated user that signed in. After signing in, the user is directed to a page, where he can choose to either transfer the guest user's assets to his personal environment, or discard them. This page uses the authenticated user's `id` to see if there are any guest users that signed in with the user's `id`.

4.2 Assets

All assets in the MS are now stored with an additional `environmentId` field to enforce multi-tenant isolation. This field ensures that each asset belongs exclusively to one environment. Aside from `environmentId`, the schema for each asset remains unchanged. All permissions checks leverage the `environmentId` field to ensure users only interact with assets in the environments to which they belong.

4.3 Folder Structure

Folders were added to the MS's assets to provide users with a tool to organize their assets and for organizations to mirror their hierarchical structures or teams. They are implemented as entries within the MS' storage solution, and processes are stored in a folder by storing a reference to the folder in the process entry.

4.3.1 Storage of Folder Structure

To realize the hierarchical folder structure within the MS, the Adjacency List model was chosen, which was discussed extensively in 3.5. This model was chosen due to its advantages in managing dynamic operations such as adding, deleting, and especially moving folders. Each folder in the MS is stored as an entry with a unique ID, a name and optionally a reference to its parent folder. Folders without a parent folder are root folders, representing the top level of the folder structure within an environment.

4.3.2 Associating Assets with Folders

Assets, specifically processes within the scope of this thesis, are also assigned to folders via a direct reference. Processes are stored with a `parentId` field, which points to the folder they belong to. This field has always got to be set, and by default it is set to the root folder of the environment the asset belongs to.

4.4 Environments

This section will cover the implementation details of environments. In its essence, an environment is just an entry in the MS' storage, where assets point to. To store these environments a new file was added to the MS' storage solution **??** that stores every environment. Every environment has an `id` and a flag named `organization` that indicates what type of environment it is. If an environment is an organization environment, it also stores information about the organization, whereas personal environments store a reference to the user that owns the environment in `ownerId`.

```
// Schema for organization environments
{
    id: string;
    organization: true;
    name: string;
    description: string;
}

// Schema for personal environments
{
    id: string;
    organization: false;
    ownerId: string;
}
```

Listing 4.4: Schemas for organization and personal environments.

4.4.1 Creation of Personal Environments

Personal environments are created when an entry for a user is created, this ensures that every user has a personal environment. Alongside the environment, a new root folder is created for the environment, where the user can store Processes, and other folders. The environment's `ownerId` references the user that created the environment, no more information is necessary, as personal environments are only accessible by one user.

4.4.2 Memberships

Memberships store a user ID and an environment ID, representing that the user is part of that environment.

4.4.3 Creation of Organization Environments

Authenticated users can create organization environments. Upon creation, the system automatically creates the two default roles: *@everyone* and *@admin*. The creator is immediately added as a member through a membership entry, and a role mapping automatically assigns them the *@admin* role. Additionally, a root folder is created to provide the initial structure for the new environment.

4.4.4 Adding Users to an Environment

Adding users to an environment is simply done by creating a membership entry, with the organization environment's ID and the user's ID. However, the inviter typically doesn't know the invitee's internal ID, so they can use the invitee's email address instead. The MS then sends an email to the invitee, with an invitation URL that contains a JWT token in a query parameter. The token contains either the invitee's email if it isn't associated with a user, or a user ID if it is. Additionally, if the inviter has the permissions for it, he can select roles for the new member. The roles' IDs are also stored in the token.

For the case where the email isn't associated with a user, when the URL is opened, the user is redirected to the sign-in page, here he has to sign in with his email address. In the sign-in URL that is sent to him, the invitation URL is added in the `callbackUrl` query parameter. After he signs in, with the sign-in URL, he will be redirected to the invitation URL, and his email address will be associated to an existing user.

The invitation URL directs the user to the MS, where the token is verified. If the email or ID stored in the token, match the user that opened the link, then the user can decide whether to accept the invitation or not. If he does, a membership entry is created with the environment's ID and the user's ID. Furthermore, if any roles were included in the token, corresponding role mappings are created.

4.4.5 Environment Selection

Every view was modified so that its URL path begins with the selected environment ID, thus ensuring the selected environment is always reflected in the URL in the following format:
`https://staging.proceed-labs.org/<environment id>/....`
Any view a user accesses, as long as the URL path begins with an environment ID, pertains exclusively to that specific environment. The MS reads the environment ID from the URL, verifies that the user is both a member of that environment and has the necessary permissions, and then uses the environment ID to query the corresponding assets.

Users can switch between environments by selecting them from a dropdown menu, that is placed on the navigation bar.

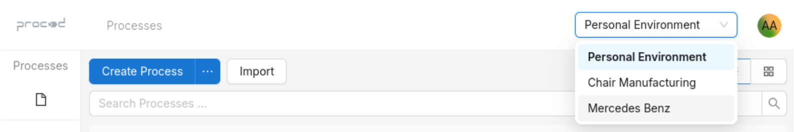

Figure 4.2: Selection of environments in the MS' navigation bar.

4.5 Roles

Each role is now directly associated with an environment by storing an `environmentId` and its permissions only apply to assets in that environment. Additionally, roles can now be associated to a folder, with the field `folderId`.

4.5.1 New Resources

Folders, environments and memberships were added to the MS' resources, and to the schema of roles, to allow permissions to be set for them. This way abilities can perform checks on these resources and enforce permissions like described in 2.3.

4.5.2 Enforcing Roles in the MS

Every action that a user can perform in an environment, is tied with a permission. For this reason, every endpoint in the MS has to check permissions, fortunately, roles were already implemented in the MS and every endpoint was already using abilities to check permissions. Abilities are built with the permissions stored in roles that a user has. Every endpoint receives an environment ID as an argument, and uses the roles the user has in that environment for building his ability. Additionally, the abilities used to check permissions are built with an environment ID, and they check the `environmentId` field of every asset to ensure that an ability from one environment can't be used to access assets from another environment.

To simplify the process of obtaining a user's ability within a specific environment, the `getCurrentEnvironment` function has been implemented. This function takes an environment ID as input and leverages the authentication information provided by NextAuth.js to identify the calling user. It then verifies the user's membership in the environment and constructs an ability based on the user's assigned roles within that environment.

```
async function createProcess(processValues, environmentId) {
  const { ability } = await getCurrentEnvironment(spaceId);

  if (!ability.can('create', toCaslResource('Process', newProcess))) {
    return userError('Not allowed to create this process');
  }

  ...
}
```

Listing 4.5: Example of an endpoint using an ability.

4.5.3 Checking Permissions associated to a Folder

A simple representation of a folder structure is computed in the MS and stored in a JSON serializable object. Each key is a folder id, and its value is its parent's id. Only the root folder's ID isn't contained in the object, as it has no parent. This structure allows traversing the tree upwards and get a folder's ancestors. With the help of this structure, two new conditions were added to abilities:

- `$property_has_to_be_descendant_of`: checks if a resource is a child of a folder by going up the folder structure.
- `$property_has_to_be_ancestor_of`: checks if a folder is a parent of a hardcoded folder, by starting at the hardcoded folder and going upwards through the tree sructure. This condition will only be applied it the asset that is being checked is a folder.

When a role is being used to build an ability, each permission for each asset is turned into a rule. When a role is associated to a folder, the rules for processes and folders, use the `$property_has_to_be_descendant_of` condition, so that the permissions only apply to assets that are descendants of the folder. This way the permissions cascade down the folder structure. Additionally, if the role allows a user to view, either a folder or a process, a rule is added, that allows the user to view all ancestors of the asset's folder. This is done by creating a rule for folders with the `$property_has_to_be_descendant_of` condition set to the asset's folder.

5 Evaluation

In this chapter, we evaluate the implementation of environments in the PROCEED MS by reviewing the requirements defined in section 1.2. The primary goals were to introduce isolated workspaces for both personal and organizational use, enable efficient asset management through a hierarchical folder structure, and adapt the role-based access control system to support these new workspaces. Table 5.1 provides an overview of the task statuses:

Task	Status
1. Implement environments	Partially implemented: folders were only implemented for processes.
2. Personal and organization Environments	Implemented
3. Adapt the MS' user management system to fit environments	Implemented
4. Adapt the MS' role system to fit environments	Implemented

Figure 5.1: Evaluation of Task List.

According to **task 1** Environments are stored as entries in the MS' storage solution with a unique ID. Every asset in the MS explicitly stores the ID of the environment it belongs to. Memberships to environments are stored in a separate table, where each entry has a user ID and an environment ID, later on we will explain how the access to assets is managed. Furthermore, each environment has a root folder, which contains more folders and processes. Apart from root folders, both process and folders store the ID of the folder they belong to.

According to **task 2** environments store a flag to determine whether they are a personal or an organization environment. Personal environments are created when a user is created and organization environments can be created by signed-in users. Personal environments have a restricted feature-set, this is enforced by the role system, this will be explained in more detail when we address task 4. Users in organization environments can potentially use all the MS' features, this is determined by the roles they have inside that environment. The user that creates an organization environment is assigned the role of admin inside that environment. Access to assets in organization environments is managed by the role system described in task 4, the admin role has all permissions for all assets in the environment. Organization environments can have multiple members, new members can be invited by other members with the right permissions.

According to **task 3**, the user management was redesigned to accommodate multi-tenancy, users are now stored as records in the MS' storage solution independently of the environments they belong to. This allows for them to be part of multiple environments. Users also have the option to sign in as a guest, which allows them to try the MS inside a personal environment without having to sign up. Guest users are stored as other users, but with a flag that indicates that they are a guest.

According to **task 4**, the MS' role system was adapted to fit environments and folders. In its essence, roles work as they do before with two key differences: Each role belongs to an environment and is only applied for assets inside that environment, and roles can be scoped to folders, meaning that its permissions apply on all descendants of the folder. The role system was also leveraged to restrict the feature set of personal environments: while personal environments don't have roles, the MS still uses the role system to manage access to assets inside personal environments, it gives the owner of the environment all permissions for the allowed features and restricts the rest.

1.2 also describes a set of non-functional requirements mostly related to developer experience. In contrast to functional requirements, which can be evaluated by determining weather the task was completed or not, non-functional requirements are harder to evaluate, as they are subjective. For this reason we determined that surveying the developers that are working on the MS would be the best way to evaluate these non-functional requirements. The survey contained six questions, each question was meant to evaluate one of the non-functional requirements. Most questions were answered with a scale from 1 to 5, for some questions 1 meant that the task was perfectly achieved and for others it was the opposite. For these questions the task is considered as sufficiently met if the average of the answers tends to the favorable side.

Non-functional requirement 1 was to keep the changes to the MS to a minimum, to evaluate this the survey asked: "On a scale of 1 (very minimal) to 5 (extensive), how significantly do you feel the codebase had to change to support multi-tenant environments?". Figure 5.2 shows a histogram of the answers to this question. The answers lie mostly in the middle, with a slight tendency towards the lower end of the scale, thus we conclude that this task was sufficiently met.

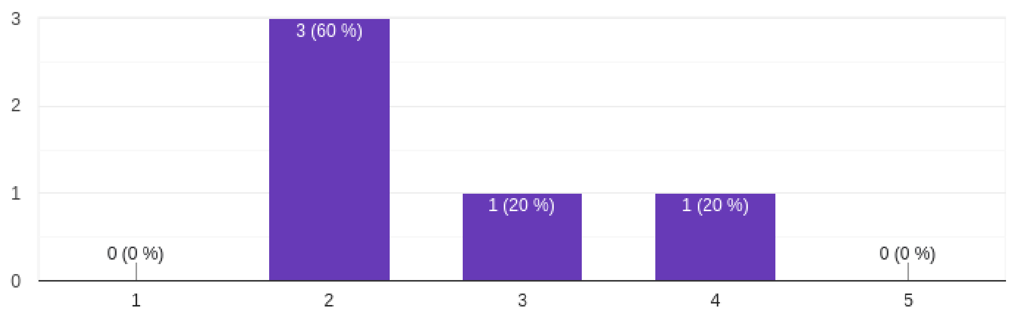

Figure 5.2: Histogram of answers to the question: "On a scale of 1 (very minimal) to 5 (extensive), how significantly do you feel the codebase had to change to support multi-tenant environments?"

Non-functional requirement 2 was to provide an intuitive user interface, to evaluate this the survey asked: "On a scale of 1 (not intuitive at all) to 5 (extremely intuitive), how would you rate the ease of navigating and managing folders/environments in the new UI?". Figure 5.3 shows a histogram of the answers to this question. The answers lie mostly in the upper end of the scale, thus we conclude that the user interface is sufficiently intuitive.

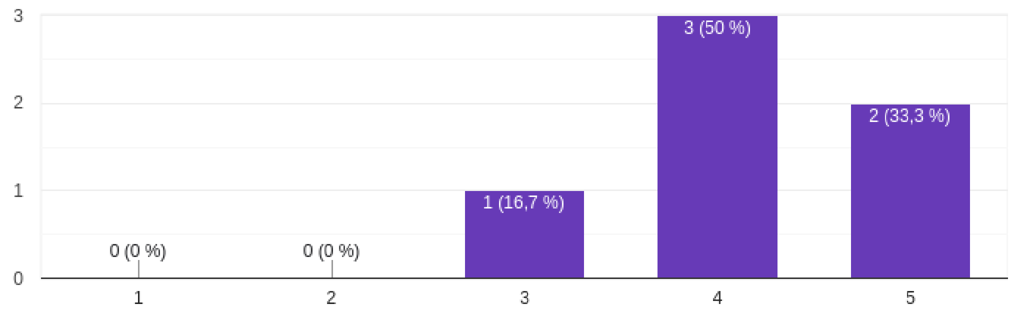

Figure 5.3: Histogram of answers to the question: "On a scale of 1 (not intuitive at all) to 5 (extremely intuitive), how would you rate the ease of navigating and managing folders/environments in the new UI?"

Non-functional requirement 3.a was that personal and organization environments share the same data structures and functions, to improve the developer experience. To evaluate this the survey asked a yes or no question: "Do you feel that personal and organization environments share sufficiently unified structures and functions to use them?". The six developers that answered this question answered "yes", thus we conclude that this task requirement was met.

Non-functional requirement 3.b was to choose a simple data structure for the folder system and to provide simple functions to manage it., To evaluate this the survey asked: "On a scale of 1 (very simple) to 5 (overly complex), how would you rate the complexity of the folder system's data model and its related functions?" Figure 5.4 shows a histogram of the answers to this question. Four people answered "2" and two people answered "3", thus we conclude the user interface is sufficiently intuitive.

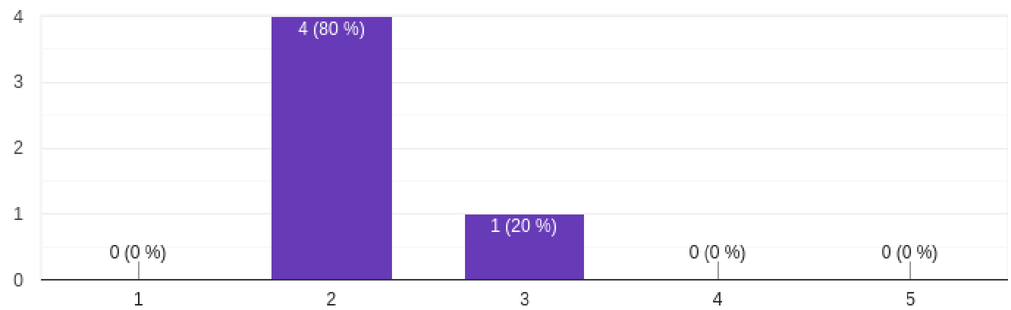

Figure 5.4: Histogram of answers to the question: "On a scale of 1 (very simple) to 5 (overly complex), how would you rate the complexity of the folder system's data model and its related functions?"

Non-functional task 3.c was to streamline environment identification and permissions check in the backend. To evaluate this the survey asked: "On a scale of 1 (not at all effective) to 5 (extremely effective), how effective are the new backend helper functions in streamlining environment identification and permission checks?" Figure 5.5 shows a histogram of the answers to this question. One person voted "3" and the rest of votes were in the favorable side of the scale, thus we conclude that this task was sufficiently met.

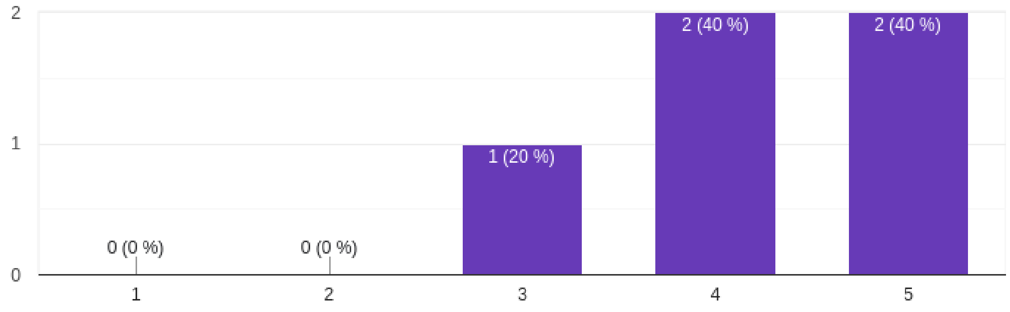

Figure 5.5: Histogram of answers to the question: "On a scale of 1 (not at all effective) to 5 (extremely effective), how effective are the new backend helper functions in streamlining environment identification and permission checks?"

Non-functional task 3.d was to introduce simpler helper functions for the frontend, to adapt the interface to a user's permissions within an environment. To evaluate this the survey asked: "On a scale of 1 (no change in complexity) to 5 (drastically simpler), how much do the new frontend helper functions simplify adapting UI elements based on each user's permissions?" Figure 5.6 shows a histogram of the answers to this question. One person voted "3" and the rest of votes were in the favorable side of the scale, thus we conclude that this task was sufficiently met.

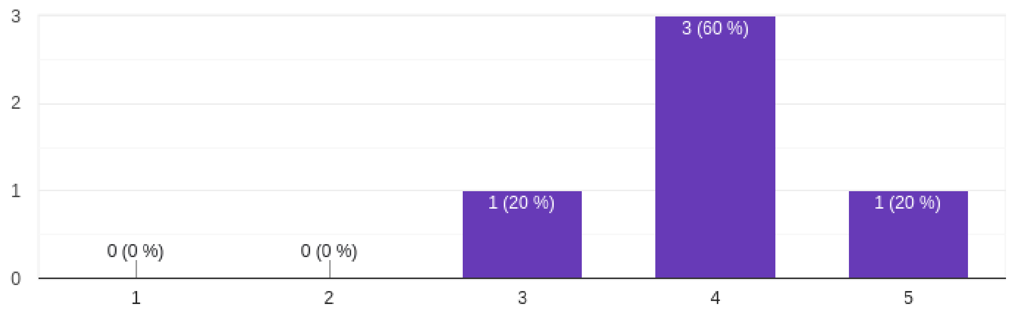

Figure 5.6: Histogram of answers to the question: "On a scale of 1 (no change in complexity) to 5 (drastically simpler), how much do the new frontend helper functions simplify adapting UI elements based on each user's permissions?"

6 Conclusion

Building an application structure that supports multi-tenancy presents many challenges, as demonstrated throughout this thesis. In addressing these challenges within the PROCEED Management System (MS), we introduced the concept of environments, isolated workspaces that support both individual and collaborative use cases.

Significant architectural adjustments were required, particularly in restructuring how assets are managed. Assets were adapted to explicitly associate them with specific environments, ensuring clear ownership and secure isolation. To enhance organization and reflect user hierarchies more accurately, a flexible folder structure was implemented. The preexisting role system was adapted to respect environment boundaries and folder structures, ensuring isolation and further enhancing the ability to represent an organization's hierarchy.

Key challenges included restructuring the MS without disrupting existing functionality, ensuring robust privacy and isolation between environments, and adapting the role system to account for both folder and environment-specific permissions. Moreover, the introduction of guest users added complexity to the authentication flows.

6.1 Outlook

While most outlined objectives were successfully achieved, there remain opportunities for future work, such as extending the folder structure to support additional asset types.

Nevertheless, this thesis establishes a robust foundation for multi-tenancy in the PROCEED MS allowing numerous additional features to be explored to further enhance user experience and productivity. For example enabling real-time collaboration on assets such as processes.

Additionally, introducing user directives would significantly simplify user and organization management by enabling automated account creation and streamlined onboarding processes.

List of Tables

List of Figures

Bibliography

[1] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6749

[2] N. Sakimura, J. Bradley, and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients," RFC 7636, Sep. 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7636

[3] C.-P. Bezemer and A. Zaidman, "Multi-tenant saas applications: maintenance dream or nightmare?" in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 88–92. [Online]. Available: https://doi.org/10.1145/1862372.1862393

[4] S. Pushpan, "Multi-tenant architecture: A comprehensive framework for building scalable saas applications," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 10, no. 6, pp. 1117–1126, 2024. [Online]. Available: https://ijsrcseit.com/index.php/home/article/view/CSEIT241061151/CSEIT241061151

[5] J. Celko, *Joe Celko's Trees and Hierarchies in SQL for Smarties, (The Morgan Kaufmann Series in Data Management Systems)*, 1st ed. Morgan Kaufmann, 2004. [Online]. Available: http://www.amazon.com/Hierarchies-Smarties-Kaufmann-Management-Systems/dp/1558609202

Appendices

Appendix 1

```
for($i=1; $i<123; $i++)
{
    echo "work harder! ;)";
}
```