Multi-Tenancy in Cloud Applications 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

October 9, 2024

Supervised by:
Prof. Dr. Axel Küpper

Assistant supervisor:
Kai Grünert

Eidestattliche Erklärung / Statutory Declaration

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

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, October 9, 2024 Chuck Norris' son

Abstract

In this thesis, we show that lorem ipsum dolor sit amet.

Zusammenfassung

Hier kommt das deutsche Abstract hin. Wie das geht, kann man wie immer auf Wikipedia nachlesen `http://de.wikipedia.org/wiki/Abstract`...

Contents

1 Introduction

In today's digital age, businesses heavily rely on cloud applications, software tools that are accessed and run entirely over the internet. These tools represent a paradigm shift from traditional software applications, where the majority of the workload happened on the user's device. Shifting some of the workload to the cloud offers many advantages:

- Accessibility: They can be accessed anywhere from anywhere with an internet connection.
- Cost-Efficient: Most cloud applications implement a payment structure, where users pay based on how much they use the application.
- Collaboration: Typically, collaboration is easier since everything can be found in one place, instead of having to send files back and forth.
- Data safety: All files are stored by the application in the cloud, and they don't have to be stored in the user's device, which could be lost, stolen or damaged.
- Device agnostic: many cloud applications can be accessed through different device types.
- No IT overhead: users don't have to setup the application on their own, which would require technical knowledge.

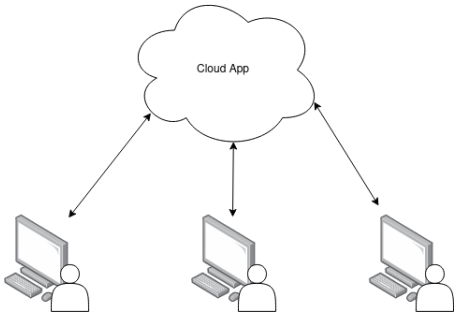

Figure 1.1: Users can access cloud applications from any device with an internet connection.

One very common feature that makes these benefits possible is called "multi-tenancy". Multi-tenancy is a software architecture in which one single instance of an application can be used by many different users or organizations at the same time. Without multi-tenancy, each user or organization would need to run the application on their own servers or computers, largely

negating the numerous benefits listed earlier.

Think of it like a big apartment building. Each tenant (user or organization) has their own private space (their assets), but they're all using the same building (the cloud application).

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.

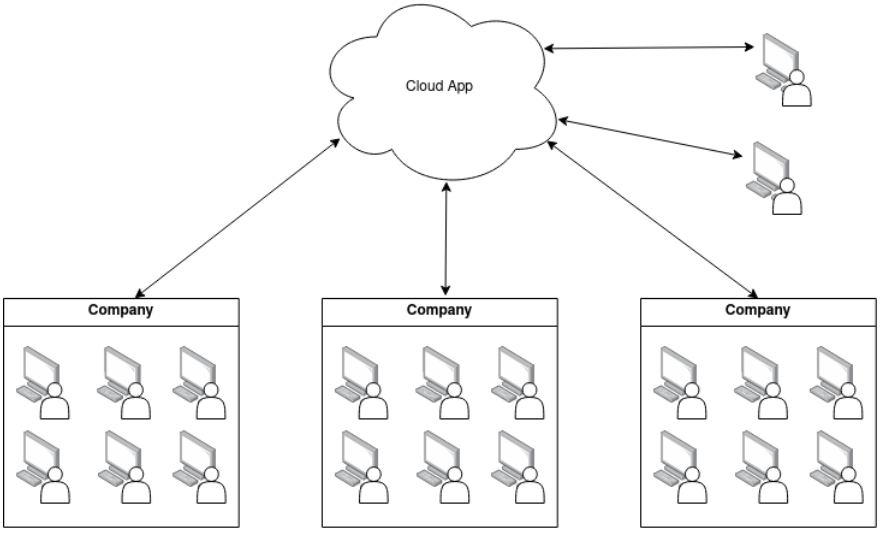

Figure 1.2: Multi tenancy in cloud applications: the same instance of the cloud application, can be used by different tenants, with different structures, without them knowing about each other.

PROCEEDis a Business Process Management System. PROCEED uses BPMN at its core to model and execute business processes. BPMN (Business Process Model and Notation) is a standardized graphical notation used for documenting business processes. BPMN is typically used inside of organizations to illustrate sequences of tasks, decision points, and interactions within various business processes, providing a standardized visual representation.

PROCEED offers two products:

- Distributed Process Engine (DPE for short): the DPEs execute BPMN processes.
- Management System (MS for short): the MS is a cloud application that gives users a graphical interface to work on their BPMN processes and deploy these to the DPEs.

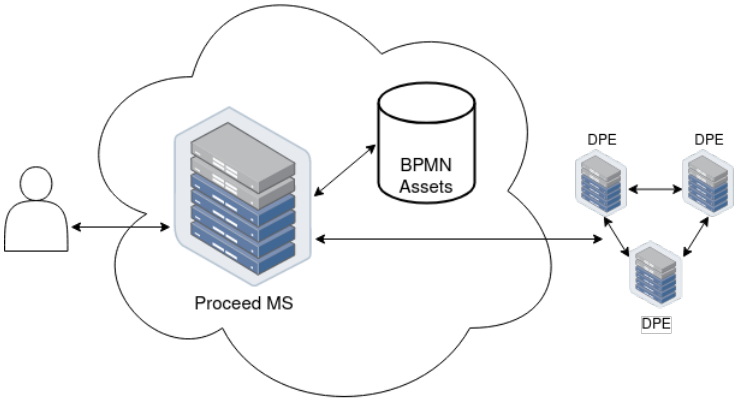

Figure 1.3: Overview of PROCEED: users interact with the Management System, to create and manage BPMN models. The Management System has the ability to deploy these models to the Distributed Process Engines.

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, PROCEED only supports universal sharing, meaning that all users would be able to see the shared assets. Furthermore, even if it was possible to share assets only to a group of users, it would be very cumbersome and error-prone.

For this reason, this thesis implements multi-tenant functionality into the PROCEED MS by introducing the concept of Environments. The MS should be able to hold multiple isolated environments, where tenants can work on their assets. Each user, who signs in, should automatically have their own Environment, this allows users to work on personal projects. Organizations will be able to create environments where multiple members can work together. Additionally, environments should include a folder structure to improve the organization of assets.

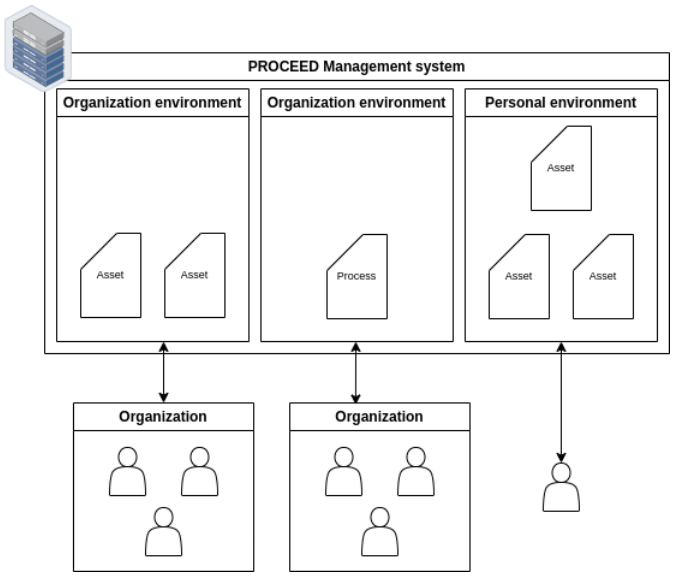

Figure 1.4: Goal of this thesis: tenants can work in their own isolated environments in the PROCEED Management System.

2 Research Questions

1. Environment Representation: How can we model environments within the existing PROCEED database schema 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 and consistent data model.
   - Efficient: find a schema that allows to efficiently query the database.
2. How can we extend the existing PROCEED role system to incorporate environment-specific permissions?
3. How can organizations model their hierarchical role structures within environments to allow members of the organization to have different levels of access to assets?
4. How can the Management System allow users to manage assets on different environments?
5. How can we minimize the impact on existing code while ensuring seamless integration?

3 Task List

1 Functional

- I The MS has to support two types of environments: personal and organization environments.
  - A Every asset in the MS must be stored in one and only one environment.
  - B Assets stored in one environment can only be accessed by members of that environment.
  - C Every user has to have a personal environment, of which only he can be a member.
  - D Organization environments must be able to have multiple members and roles that control what each member can do.
  - E Environments must have a folder system to store assets.
    - i Find a suitable abstraction to represent folders in a database.
    - ii Ensure privacy between environments.
    - iii The MS's preexisting role system must be adapted to fit environments: The PROCEED Management System already has a role system in place to manage user's access to resources, these roles need to be modified for them to work with environments.
      - a Find a suitable inheritance model for roles based on the folder structure of an environment (e.g. a user with a role in a parent folder, can perform actions in all subfolders).
      - b Ensure roles are always enforced in the backend.
      - c The frontend UI must adapt to a user's roles, by only showing options that the user has permission to do.
- II The MS must be able to hold multiple environments and let users access them concurrently.
- III Users must be able to be members of multiple environments and carry out actions in each one of them.

2 Non functional

I keep changes to the existing codebase to a minimum.

II The same data structure should be used for both personal and organization environments.

III The user interface for navigating and managing folders and environments should be intuitive and easy to use.

IV Prioritize developer experience by creating clear abstractions and APIs.

- A Create simple abstractions for the backend code of the MS, that allow to acknowledge a user's environment with minimal effort.
- B Create a simple abstraction for the frontend, that facilitates adapting the Interface for each.

4 Related Work

4.1 OAuth 2.0 and OpenID Connect

Oauth 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 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 adresses 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. Insetead of handing his crendentials to the third party application, the resource owner signs in, in the application's website which then issues an access token to the client application. This method avoids the user having to share his credentials with third party applications.

4.1.1 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 succesfully authenticated.

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

4.1.2 Authorization Grants

Authorization Grants are credentials that are issued to clients, which can be exchanged for an access token. This access token can be used to access the protected resources on the resource server. Oauth 2.0 defines four authorization grants with different flows.

4.1.2.1 Implicit

The implicit grant is very helpful for public clients, as it doesn't require confidential client credentials. This is very helpful for browser-based clients, as they can't store confidential credentials securely. In the implicit grant users are redirected to the authorization server, where they authenticate thmeselves and authorize the client. Afterwhich the authorization server issues an access token directly to the client, this is done so with a HTTP redirect, where the access token is embedded in the redirect URL, this way the client can extract the access token from the URL.

In this flow the resource owner only authenticates with the authorization server, thus never having to share his credentials with the client.

Implicit grants have many security risks, as the access token is exposed in the URL and can be intercepted by a malicious attacker. This is why PKCE (Proof Key for Code Exchange) was later introduced as an addition to the implicit grant [**?**].

4.1.2.2 Resource Owner Password Credentials

This grant type requires the resource owner to share his password credentials with the client. The resource owner's password credentials represent an authorization grant, which the client can exchange them for an access token. Even though this grant type requires the resource owner to share his credentials with the client, these are only used for one request and don't have to be stored.

4.1.2.3 Client Credentials

The client credentials grant is used when the client is the resource owner. Clients are typycally issued crendentials, which they can use to authenticate themselves. Clients send these credentials to the authorization server and are issued an access token.

4.1.2.4 Authorization Code

The Authorization Code grant is the most common grant type used in OAuth 2.0, it is similar to the <implicit grant 4.1.2.1, as it also uses HTTP redirects and it doesn't require the resource owner to share his credentials with the client. In the authorization code grant, the client redirects the resource owner to the authorization server. There the resource owner authenticates

himself and authorizes the client. afterwhich the authorization server redirects the resource owner back to the client with an authorization code. The client then authenticates itself with his confidential credentials on the authorization server and exchanges the authorization code for an access token. As the client needs confidential credentials, this flow is only suitable for confidential clients. The exact steps are shown in figure 4.1.

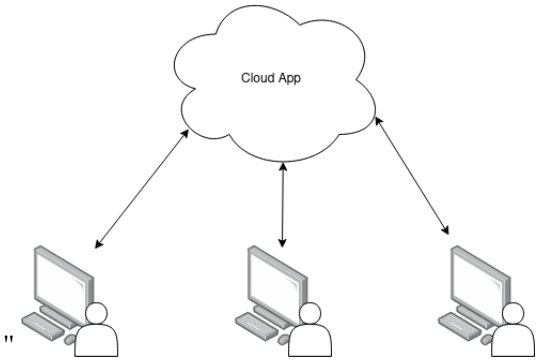

Figure 4.1: Cloud Application.

4.1.3 OpenID Connect

OpenID (OIDC for short) Connect is an identity layer built on top of the OAuth 2.0. While OAuth 2.0 focuses on authorization (granting clients access to resources), OIDC extends this to authentication. Since OIDC deals with authentication, I will call the resource owner the user from now on.

OIDC uses as its base either the Authorization Code flow 4.1.2.4 or the Implicit flow 4.1.2.1 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 identifier, which is a unique identifier for the user in the <- finish here The Id Token is sent alongside the Access token to the client, on which step this happens depends on weather the Authorization Code flow or the Implicit flow is used.

In the OIDC flow, after the client obtains the Authorization Code, it exchanges it for both an Access Token (as in OAuth 2.0) and an ID Token. The client can then validate the ID Token to ensure it's genuine and extract the user information contained within.

In essence, OIDC allows clients to obtain information about the authenticated user in a standardized way.

4.2 PROCEED's Assets

Assets are Objects that users can create and manage through the MS's interface. 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 helps with managing access to assets. Currently, the MS supports the following assets:

1. Processes, Project and Templates: These objects store BPNN at their core.

2. Task:
3. Machine: Object that represents a server running Distributed Process Engine. The machine is used to manage properties of the server.
4. Execution: An execution represents a process that is being executed distributedly.

Furthermore, the MS implements assets that are used to manage how users can use the MS, called management assets:

- Role4.3: 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.

4.3 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 they can be assigned to multiple users and don't change often, making them easier to manage than individual permissions.

4.3.1 MS's Role System Terminology

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

- Resource: A resource is any protected entity in the management system, that can be accessed by users. Resources can be assets 4.2, but they don't have to be.
- Action: An action is a specific operation that can be performed on a resource.
- Permission: A permission is a tuple of resource type and action, which specifies that a user can perform the action 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 action.
- 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.

4.3.2 MS's resources and actions

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

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

4.3.3 MS's roles in CASL

The PROCEED MS uses [1]CASL to implement Rules. 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 defined by four parameters: user action, subject, fields, conditions. User actions and subjects are analogous to actions and resources 4.3.1.

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".

Fields are used to specify which fields of a resource instance an action can be performed on, e.g. a user can update a process's name, but not its id, or creation date.

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.

```
1  import { defineAbility } from '@casl/ability';
2
3  class User {
4    constructor(id) {
5      this.id = id;
6    }
7  }
8
9  class Process {
10   constructor(user, name) {
11     this.authorId = user.id;
12     this.createdOn = new Date();
13     this.name = name
14   }
15 }
16
17 function abilityForUser(user){
18   return defineAbility((can, cannot) => {
19     can('delete', 'User', {id: user.id});
20
21     can('update', 'Process', ['name'], {authorId: user.id});
22   });
23 }
24
25 const user1 = new User(1);
26 const user1Ability = abilityForUser(user1);
27 const user1Process = new Process(user1, 'some process');
28
29 const user2 = new User(2);
30 const user2Ability = abilityForUser(user2);
31
32 user1Ability.can('update', 'Process'); // true
33 user1Ability.can('update', user1Process, 'name'); // true
34 user1Ability.can('update', user1Process, 'createdOn'); // false
35
```

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

```
36 user1Ability.can('delete', user1); // true
37 user1Ability.can('delete', user2); // false
38
39 user2Ability.can('update', 'Process'); // true
40 user2Ability.can('update', user1Process); //false
```

Listing 4.1: CASL example

If there exist any possible resource instance, where the 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 .

5 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 permissions 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.

5.1 Modifications to Assets and Resources

This thesis will modify the assets 4.2 and the resources 4.3.2 supported by the MS. Assets 5.4.4 and Resources 5.5 will be modified to be contained inside environments. As will be explained in 5.2, 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's assets and resources. Furthermore, folders 5.3 and memberships 5.4.3 will be added to the MS' assets and resources.

5.2 Users

Previously users in the MS represented a member 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 now represent individual people utilizing PROCEED.

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

All users have a personal environment 5.4.1 in which they can create and manage assets freely. Authenticated users can also be part of and create organization environments 5.4.2, where they can collaborate with other users.

5.2.1 Authenticated Users and Accounts

To allow the same user to be able to sign in with different Oauth2 providers, e.g. with Google, Facebook or Discord, 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. This way, when a user is signing in with credentials from an Oauth2 provider, the MS can look up the corresponding account to the credentials, and then find the user that the account is linked to.

5.2.2 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. 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 merge, 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.

5.2.3 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's database, alongside the data of authenticated users. Storing the data locally has two great 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's 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's codebase and it also makes it harder to use coding assistance features of IDEs like Goto definition [1]. For this reason, storing guest user's data locally isn't worth the benefits. So we decided to store a guest user's 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 he is 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.

5.2.4 Development Users

When developing the MS, it is not common for the developer to have all the necessary keys configured in the environment variables, for the MS to be able to authenticate users with Oauth2 4.1 or send sign-in emails. For this reason the MS implements two development users: johndoe and admin. When the MS is in development mode, i.e. the environment variable `NODE_ENV` is set to `development`, the sign-in page shows a new input field where you can enter the devel-

[1] `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_definition`

opments user's username. Both users are stored as authenticated users in the MS's database and can be used to test the MS during development.

5.3 Folders

Folders are intended to allow organizations to mirror their hierarchical structure and facilitate the organization of assets in general. Starting from a root folder, users should be able to store assets within folders and nest folders inside other folders, creating a flexible and intuitive structure. In this thesis, folders where only implemented to support processes, but they could be extended to support other types of assets like the ones described in 4.2.

5.3.1 Folder structure 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 [**?**], 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 has 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 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 bigger 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 easy and efficient, they can be found by querying for nodes whose paths start with the current node's path.
    - 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
    - Finding a node's children is easy but, not as efficient as finding its descendants, as you need to perform one additional check to ensure that nodes are direct children.
    - Moving nodes and their subtrees is inefficient, since it requires updating the path of every node in the subtree.

	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 5.1: 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. 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 a simple string comparison. 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. For those reasons, the MS will use the adjacency list model to store the folder structure.

5.3.2 Storing assets inside folders

Once the folder structure is implemented, it still is necessary to store assets inside folders. This thesis only implemented this feature for processes 4.2, however, the same principle could be applied to 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, where analogous to the adjacency list model, each process stores a reference to the folder it is stored in. This way, when moving a folder, it isn't required to update anything other than the folder. Moving an asset to another folder only requires updating the asset's reference.

5.4 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.

5.4.1 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 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 5.4.2.

5.4.2 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.

5.4.3 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.

5.4.4 Storing assets inside environments

All assets within the MS 4.2, 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 5.2.2.

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 to 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. This risk is mitigated by the fact, that the environment reference of assets is immutable.

5.4.5 Environment selection

Users will be a part of multiple environments, and they will need to be able to work on all of them. There are two ways of accomplishing this: the user can select one environment at a time, or he 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 the selected environment.

There are two methods of accomplishing environment selection: implicit and explicit. In the implicit method, the selected environment is managed internally and 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 only chose one. The implicit method has the advantage that the URLs are shorter and easier to read, however it has the disadvantage, that some links 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, allowing users to share links at the cost of longer URLs.

5.5 Roles

Roles define what actions a user can perform on an asset. Prior to the changes introduced in this thesis, roles in MS 4.3 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 restriction.

Folders allow organizations to mirror their hierarchical structure, but this wouldn't be entirely useful if roles 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 the association. The permissions of roles associated with a folder cascade down the folder structure, i.e. a role associated to a folder will also apply to all of its children. In this

thesis, the folder structure was only implemented for processes, this means that only the permissions for processes and folders will apply to the associated folder's subtree. Roles that aren't associated to a folder will continue to apply to all assets in the environment.

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, not the contents of the parent folders. This allows users to navigate the folder structure until they reach the assets they're allowed to view and manage.

5.5.1 Enforcing Permissions Based on 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, both in the MS and in the user's browser: the MS needs to know if a user is allowed to perform an action, and the UI on the user's browser needs to adapt to the user's permissions. Since both the MS and the browser need to know the folder structure often, we decided to compute and cache a representation of the folder structure of every organization environment, from which an asset is requested. The cache is invalidated whenever a folder is added, removed or moved.

5.5.2 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 all the users that are part of the organization environment. The permissions in this role start out empty, but can be modified.

6 Implementation

6.1 MS Architecture

Before diving into the implementation details of environments, it is important to understand the architecture of the MS. The MS is built using Next.js [1], a React [2] framework that allows for server-side rendering. Altough 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.

6.1.1 Data storage

The MS doesn't use a database management system, instead it stores all data in JSON files. Each file can be seen as a table in a traditional relational database. Even though this approach allows for unstructured data, the MS uses Zod [3] schemas to enforce a structure on data that is stored. 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 satisfies the schema.

```
1 import { z } from 'zod';
2
3 const UserSchema = z.object({
4   id: z.string(),
5   username: z.string(),
6   image: z.string().optional(),
7 })
8
9 // TypeScript type that satisfies the UserSchema
10 type User = {
```

[1] https://nextjs.org/
[2] https://reactjs.org/
[3] https://zod.dev/

```
    id: string;
    username: string;
    image?: string | undefined;
}
```

Listing 6.1: Example of a Zod schema and the corresponding TypeScript type.

6.2 Users

User authentication is implemented by leveraging OpenID Connect 4.1.3, with the help of NextAuth.js [4]. NextAuth.js stores a JWT token [5] cookie [6] in the user's browser, which is then parsed and verified by the MS' backend. If the JWT token is validt the user is considered authenticated. If the user couldn't be authenticated, he is redirected to the sign-in page. NextAuth.js supports many OAuth2 providers, we only have to provide a client id and secret for each provider. Additionally, NextAuth.js implements different callbacks that can be used to customize the sign-in flow.

6.2.1 Sign in flows

Should i talk about this?

6.2.2 Authenticated Users

Authenticated Users are users that sign in to the MS with personal information, be it an email or an Oauth2 account. For authenticated users we store an `id`, a flag named `isGuest`, set to false, to indicate that the user isn't a guest and personal information. This is the schema for authenticated users:

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

Listing 6.2: Schema for authenticated users.

All the personal information is optional, because depending on how the user signs in, the information might not be available. For instance, because NextAuth.js' email sign in only requires the user to input an email, authenticated users are created without a first name,

[4] https://next-auth.js.org/
[5] https://www.rfc-editor.org/rfc/rfc7519.txt
[6] https://www.rfc-editor.org/rfc/rfc7519.html

last name or username. A possible workaround would be to automatically generate these for

It is important to understand the difference between Users and Accounts. A user represent a person, while an accounts represents a method of signing in for a user. A user can have multiple accounts, but an account can only be linked to one user. To recognize an account we need to store the name of the provider and the account's id on the provider's platform. And to link the account to a User, we need to store the User's id on the MS. This is the schema for accounts:

```
{
    id: string;
    type: "oauth";
    userId: string;
    provider: string;
    providerAccountId: string;
}
```

Listing 6.3: Schema for accounts.

6.2.3 Guest Users

Users that aren't signed in can choose to try the MS out as a guest, this doesn't require the user to input any personal information.

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's database, alongside the data of authenticated users. Storing the data locally has two great 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's database. However, this approach has one key downside, the MS would have to implement two storage solutions and accordingly switch between them. The added complexity of storing guest user's data locally isn't worth the benefits, so we decided to store a guest user's data in the MS's database and create an entry for him in the Database like for all users, with the difference that a flag is set, to indicate that he is a guest. This way, all the same endpoints that authenticated users can call to interact with the MS can also be called by guest users. An important caveat is that now all relevant endpoints have to check whether the user

- user schema - accounts represent ways for a user sign in - user can have multiple accounts - account linking - sign in methods (+ dev sign in)

6.3 Assets

- environmentId stored on each thing to improve querying - talk about data normalization - talk about breaking normalization for perfomance gains -> reference a paper or smth

6.4 Environments

- environments are entry in db - memberships - environment format entry - verification (when envs are created by not signed in users) - creation - deletion - managing the env - section for folders - decide how to divide

Environments are stored as an entry in the MS table

6.4.1 Creation

6.4.2 Memberships

6.5 Roles

7 Evaluation

The evaluation of the thesis should be described in this chapter

8 Conclusion

Describe what you did here

List of Tables

List of Figures

Appendices

Appendix 1

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