# SCA Tool User, Organization, and System Management and Administration

MASTER THESIS

**Philipp Hoffmeister**

Submitted on 9 April 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Martin Wagner, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

# Declaration of Originality

I confirm that I have written this thesis unaided and without using sources other than those listed and that this thesis has never been submitted to another examination authority and accepted as part of an examination achievement, neither in this form nor in a similar form. All content that was taken from a third party either verbatim or in substance has been acknowledged as such.

---

Erlangen, 9 April 2025

# License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

---

Erlangen, 9 April 2025

# Abstract

To stay legal when publishing software with open-source components, the distributor is required to provide an accurate Software Bill of Materials (SBOM) and legal notices. While the requirement may sound simple, through the lack of affordable and simple tools, the process to generate those is generally tedious. SCA Tool's goal is to mitigate those problems by provding an easy to use tool, automating most of the process. One important aspect the tool is currently lacking is sharing resources through the concept of organizations. Consequently, during the work of this thesis, organizations are integrated into SCA Tool. To limit the access different members of an organization have on the organization's resources, an access control system is introduced. A variation of Role-based Access Control (RBAC) is implemented through Permify, with an additional wrapping- and translation-layer integrated into SCA Tool's backend. Additionally, to allow system administrators to manage users and organizations a system management dashboard is added.

# Contents


**1 Introduction** **1**

**2 Literature review** **3**

2.1 Access Control Basics . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Policies, Models and Mechanisms . . . . . . . . . . . . . . . . . . . . . 4
2.3 Access Control Policies . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3.1 Discretionary Access Control . . . . . . . . . . . . . . . . . . . 5
2.3.2 Mandatory Access Control . . . . . . . . . . . . . . . . . . . . 6
2.3.3 Role-based Access Control . . . . . . . . . . . . . . . . . . . . . 7
2.4 Microservice Database Consistency . . . . . . . . . . . . . . . . . . . 10

**3 State of Technology** **13**

3.1 Ory Keto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Casbin . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.3 Open Policy Agent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.4 OpenFGA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.5 Permify . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

**4 Requirements** **25**

4.1 Functional Requirements . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.1.1 Users . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.1.2 Addition of Organizations . . . . . . . . . . . . . . . . . . . . . 25
4.1.3 System Administration . . . . . . . . . . . . . . . . . . . . . . . 26
4.1.4 Access Control . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.2 Non-Functional Requirements . . . . . . . . . . . . . . . . . . . . . . 27
4.2.1 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.2 Portability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.3 Maintainability . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.4 Reliability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.5 Integrity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**5 Considerations** **29**

5.1 Best Fitting Policy . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2 Authorization Customizability . . . . . . . . . . . . . . . . . . . . . 30
5.3 Non Resource-bound RBAC . . . . . . . . . . . . . . . . . . . . . . 30
5.4 Access Control Implementation . . . . . . . . . . . . . . . . . . . . 34

**6 Architecture 37**
6.1 Users and Organizations . . . . . . . . . . . . . . . . . . . . . . . . 37
6.1.1 Adding Organizations . . . . . . . . . . . . . . . . . . . . 37
6.1.2 Personal Projects . . . . . . . . . . . . . . . . . . . . . . . 38
6.2 Authorization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**7 Design and Implementation 41**
7.1 Users and Organizations . . . . . . . . . . . . . . . . . . . . . . . . 41
7.1.1 Organization-Relationship Types . . . . . . . . . . . . . . 41
7.1.2 User Experience of Organization Management . . . . . . . 42
7.2 Access Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
7.2.1 Translating Non Resource-bound Roles . . . . . . . . . . . 42
7.2.2 Role Management . . . . . . . . . . . . . . . . . . . . . . 43
7.2.3 Handling different resources . . . . . . . . . . . . . . . . . 44
7.2.4 Authorization Enforcement . . . . . . . . . . . . . . . . . 47
7.2.5 Deploying Permify . . . . . . . . . . . . . . . . . . . . . . 50
7.2.6 Making Permify Accessible to the Backend . . . . . . . . . 50
7.2.7 Database Consistency . . . . . . . . . . . . . . . . . . . . 51
7.2.8 Migration of Permify Schemas . . . . . . . . . . . . . . . . 53
7.3 System Administration Dashboard . . . . . . . . . . . . . . . . . . . 54
7.3.1 Enhancing User States . . . . . . . . . . . . . . . . . . . . 54
7.3.2 Administration Dashboard . . . . . . . . . . . . . . . . . . 55

**8 Evaluation 57**
8.1 Functional Requirements . . . . . . . . . . . . . . . . . . . . . . . . 57
8.1.1 Users . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
8.1.2 Addition of Organizations . . . . . . . . . . . . . . . . . . 57
8.1.3 System Administration . . . . . . . . . . . . . . . . . . . . 57
8.1.4 Access Control . . . . . . . . . . . . . . . . . . . . . . . . 58
8.2 Non-Functional Requirements . . . . . . . . . . . . . . . . . . . . . 58
8.2.1 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . 58
8.2.2 Portability . . . . . . . . . . . . . . . . . . . . . . . . . . 59
8.2.3 Maintainability . . . . . . . . . . . . . . . . . . . . . . . . 59
8.2.4 Reliability . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
8.2.5 Integrity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

**9 Conclusions 61**

**References** **63**

# List of Figures


2.1 Authentication and Access Control during an access request . . . 4
2.2 Indirection of access through roles (Samarati & de Vimercati, 2001, p. 182) . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 Hospital example of role hierarchy (Ferraiolo et al., 1995, p. 244) 9
2.4 Example of orchestrator-based saga (Aydin & Çebi, 2022) . . . . 11
2.5 Example of choreography-based saga (Aydin & Çebi, 2022) . . . . 11

5.1 Classical RBAC, best-case example . . . . . . . . . . . . . . . . . 31
5.2 Classical RBAC, realistic example with grouped projects . . . . . 32
5.3 Classical RBAC, worst case scenario with no senseful project-groups 32
5.4 Non resource-bound approach of scenario in Figure 5.1. Stripped to only two users for visual clarity. . . . . . . . . . . . . . . . . . 33
5.5 Non resource-bound approach of scenario in Figure 5.3 . . . . . . 34

6.1 Current imlementation of organizations . . . . . . . . . . . . . . . 37
6.2 New imlementation of organizations . . . . . . . . . . . . . . . . . 38
6.3 Integration of authorization into SCA Tool . . . . . . . . . . . . . 39

7.1 Relation between role-templates and role-instances . . . . . . . . 43
7.2 Flowcharts of role-management examples . . . . . . . . . . . . . . 45
7.3 Interaction between SCA Tool's components during role-assignment of user on resource . . . . . . . . . . . . . . . . . . . . . . . . . 46
7.4 Resource interface, implemented by SCA Tool's resources . . . . . 47
7.5 Resource-request being handled by the different layers of the backend 50
7.6 Handling consistency issues during role permission updating by manually rolling back Permify's data. Rollback processes are filled red. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

# List of Tables

7.1 Available permissions on organizations . . . . . . . . . . . . . . . . 48
7.2 Available permissions on projects . . . . . . . . . . . . . . . . . . . . 48
7.3 Available permissions on code-units . . . . . . . . . . . . . . . . . . . 49
7.4 Available permissions on code-unit versions . . . . . . . . . . . . . . 49

# Acronyms

**SCA** Software Composition Analysis

**SBOM** Software Bill of Materials

**MAC** Mandatory Access Control

**DAC** Discretionary Access Control

**RBAC** Role-based Access Control

**ABAC** Attribute-based Access Control

**ACL** Access Control List

**OPL** Ory Permission Language

**OPA** Open Policy Agent

**CNCF** Cloud Native Computing Foundation

**ReBAC** Relationship-based Access Control

**SDK** Software Development Kit

**DSL** Domain-specific language

**JSON** JavaScript Object Notation

**ADM** Anemic Domain Model

**RDM** Rich Domain Model

**OOP** Object-oriented programming

**UX** User Experience

# 1 Introduction

SCA Tool[1] is an online tool, trying to make the distribution of software with open-source components easy, by helping to fullfill the legal obligations that come with distributing software. It offers various functions, mainly SBOM management, governance, license compliance and vulnerability management and currently focuses on JavaScript and npm. While alternatives like Black Duck[2], FOSSA[3] and FOSSology[4] offer similar features, they often either entail high cost or setup- and usage complexity, if not both. SCA Tool tries to counter this, by allowing users to provide as little or as much input as desired and generate the best possible output for all cases. Users who simply want a good enough SBOM or legal notices can make use of SCA Tools automations and only need to provide the source code. If the results are not sufficient to the user, they can give further input to optimize the result in quality. For users who are willing to give high amounts of input and corrections, currently developed features like curation allows them to adjust the results to their needs. SCA Tool consequently is suitable for a variety of users with different requirements.

One aspect that SCA Tool is currently lacking is the concept of shared resources, or more precisely for multiple users to have access to the same resources. At the moment, each SCA Tool user are solely able to create personal projects for them to work on. While this can be sufficient for single developers, it makes SCA Tool undesirable for teams or organizations. In the scope of this thesis, this problem shall be solved.

The concept of organizations as a group of users shall be introduced to SCA Tool. Resources are no longer owned by users but by organizations, potentially allowing all members of such an organization to access them. In most scenarios, members of an organization have separate responsibilities on different resources. To correctly capture such separation of duties, it is desirable for the users to only have the necessary access to fullful their responsibilities. Consequently, an access

[1] https://scatool.com/
[2] https://www.blackduck.com/
[3] https://fossa.com/
[4] https://www.fossology.org/

control (or authorization) system shall be integrated into SCA Tool. Finally, to allow system administrators to manage and moderate all users and organizations, a system management dashboard needs to be added to the application.

The thesis is split into nine chapters. After the introduction the currently available information on access control is presented in the form of literature review and a presentation of currently available access control tools. Afterwards, the requirements for the thesis are defined and considerations on possible solutions are presented. The architecture, design and implementation is explained in detail in the following. In the end the previously declared requirements are evaluated and conclusions are made.

# 2 Literature review

The addition of organizations to SCA Tool implies that multiple users, the members of an organization, will have shared access to the resources of an organization. In a perfect world each user would be trustworthy enough to have full access to all resources in an organization, but in a more practical setting users should be limited in the access they have. This might for example entail users not being able to perform some actions on a resource. In the context of SCA Tool, one such case could be that not every member of an organization should be able to edit or even delete said organization. Furthermore there might be different teams in an organization, each having different projects or aspects they are responsible for. Consequently it would be beneficial to limit their access to exacly these subsets of resources. This might be a subset of projects in SCA Tool or even just certain areas like license compliance or vulnerability management. As described by Hu et al. (2006, pp. 3-4), for most cases this can be formally defined as a *subject* performing an *operation* on an *object*. A subject is defined as the executor of an operation, usually the user trying to access some resource (the object). The kind of action the subject is trying to execute on the object, implying the kind of access that is needed, is the operation. This concept of limiting access usually is referred to as Access Control.

## 2.1 Access Control Basics

As explained by Sandhu and Samarati (1994, p. 40), Access Control is an additional security filter with the intention of being combined with other security mechanisms of a system such as authentication. While authentication ensures the correct identity of a user, usually being enforced by a user having to log in to the system, Access Control is responsible for authorizing the access to resources (see Figure 2.1).

But, as they state, authorization by itself is not sufficient for a full implementation of Access Control. Additionally the concept of auditing should be introduced. Auditing describes the concept of an adminstative figure being able to trace all interactions users of a system have performed on said system. More practically

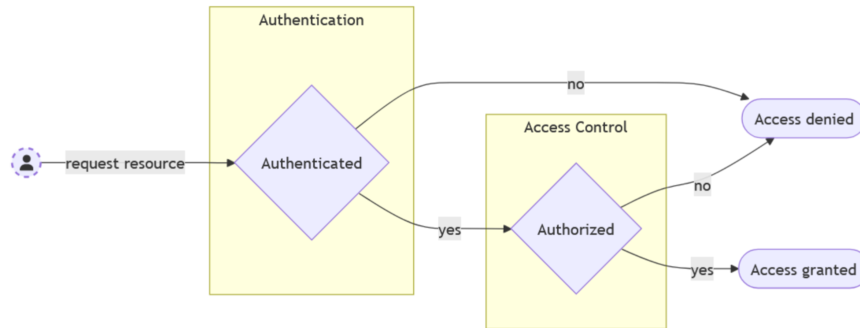



**Figure 2.1:** Authentication and Access Control during an access request

this usually entails the logging of user operations and the possibility for authorized adminstators to later review these. In the context of Access Control this boils down to the information of which users tried to access which resources in what manner and if this request was deemed as authorized by the system. One promiment reason to add auditing, displayed by Sandhu and Samarati (1994, p. 40), is the added transparency to the administrators. Misconfigurations or even software bugs, causing theoretically unauthorized access may be found, analyzed and resolved continously, leading to a more robust authorization of a system.

## 2.2 Policies, Models and Mechanisms

As explained by Samarati and de Vimercati (2001, pp. 137-138), the concept of Access Control can generally be broken down into three abstraction layers: policy, model and mechanism. A *policy* is the most abstract level and describes the rough structure of how authorization shall be enforced, often in the form of rules. There are noticeably different kinds of Access Control policies, which shall be looked into later on. The noticeably different kinds of access control policies are discussed further in Section 2.3. The next abstraction layer is referred to as a *model* and may be described as the formalization of the abstract rules defined by a policy. It practically contains the information of the policy but expressed in a way that is easier for the implementation to work with. A *mechanism* is the least abstract level and describes the actual implementation of how the previously defined rules are enforced. It should be noted that while a mechanism describes the implementation of one policy in this context, a mechanism might be capable of implementing different policies. Concluding, the separation of the rather complex concept of Access Control in layers allows easier development. This is not only since the different layers may be developed individually, but also through the explicit separation of concerns.

## 2.3 Access Control Policies

While the general idea of Access Control policies was already discussed, it is worth looking deeper into the different kinds. Like Hu et al. (2006, p. 5) explains, it is not productive to simply present an exhaustive list of policies, not only since policies continously change over time, but also since there are numerous variants for individual use cases. Rather it is worth looking into the different types of policies and what differentiates them. Sandhu and Samarati (1994, p. 44), Samarati and de Vimercati (2001, p. 139) and Hu et al. (2006, pp. 5-10) categorize policies into:

- Discretionary Access Control (DAC)
- Mandatory Access Control (MAC)
- Role-based Access Control (RBAC)

The following subsections describe the characteristics of policies in these three categories.

### 2.3.1 Discretionary Access Control

As Sandhu and Samarati (1994, p. 44) and Samarati and de Vimercati (2001) state, discretionary policies base access control on the user trying to access a resource and specified authorization rules. More specifically the policies determine if some subject has the needed access, either in the form of a permission or as access mode, on some object. Sandhu and Samarati (1994, p. 44) furthermore elaborates that most often discretionary policies are *closed* (in comparison to being *open*), meaning that if no authorization rule exists that explicitly gives the subject the needed permission, then the request is denied.

Hu et al. (2006, p. 6) explains that the aspect that separates discretionary from non-discretionary access control is based on who is capable of defining or modifying authorization. In non-discretionary access control authorization rules, and therefore which subjects have access to which objects is solely determined by a central administrative figure. In the context of Discretionary Access Control (DAC) on the other hand this does not have not be the case. They explain that with the concept of ownership of an object, access to said object may for example be given to other users by its owner, in comparison to a central administration. The exact nature of authorization administation is not fixed to this basic concept though, and as Samarati and de Vimercati (2001, p. 174-175) show, a variety of *adminstative policies* may be used in DAC. They present the following possibilities:

- **Ownership:** Every object is owned by a user. Said owner may authorize

other users access on their owned objects.

- **Decentralized:** Based on the concept of **Ownership**, with the addition that not only owners may grant other users access, but that owners are also able to delegate their authorization capabilities to other users.
- **Centralized:** Central authorization adminstration is executed by one user or one group of users.
- **Hierarchical:** Similar to **Centralized** in the aspect that a group of users are responsible for administrating authorization. Differentiates in the fact that said administrators are assigned by a central administrator one level above them in the hierarchy.
- **Cooperative:** Access cannot be given by a single administrative user. Instead multiple authorizers need to allow the access.

Consequently, through the freedom in defining suitable authorization rules and the variety of usable adminstative policies, discretionary policies are highly flexible and therefore suitable for a lot of real world applications, as stated by Hu et al. (2006, p. 6). It should be noted though, that it is directly bound to a few downsides, often mentioned in literature. As elaborated by Hu et al. (2006, p. 6), Sandhu and Samarati (1994, p. 44) and Samarati and de Vimercati (2001, p. 146-148) DAC does not restrict the flow of information and does not strictly separate the concept of user and subject. Since the flow of information is unrestricted, even though some user might only have viewing permissions on a resource, it is not enforced that said information is not passed on to an unauthorized user. Additionally the missing separation between an actual user accessing a resource or a subject, which could also refer to a process running on behalf of the user, DAC would allow programs, including malicious code, to run with the full set of permissions of the executing user.

### 2.3.2 Mandatory Access Control

The main difference between discretionary policies and mandatory policies is the aspect of flexbility in authorization administration. As previously described DAC tends to be highly flexbile in that regard, bringing respective positives, but also some security risks. As explained by Samarati and de Vimercati (2001, p. 148 - 151) this is different for mandatory policies. They base their authorization management, so who is able to give users access to resources, solely on a centralized concept, similar to centralized administrative policies in DAC. Consequently one single entity is responsible for managing access of users of the system.

They explain that the likely most popular example for a mandatory policies are *multilevel security policies*, which simplified, use classification to model access

control. A set of classes if defined, with each having some security level, determining how secure objects of this class have to be treated. Through this security level it is possible to model an order in which some objects are seen as more secure than others. To now determine which subjects have access to which classes of objects, subjects are classified as well. Subjects with a class of the same or a higher security level as some resource are then authorized to access said resource. An example they provide is a military applications with the following categories ordered by their security level:

$$\text{Top Secret} > \text{Secret} > \text{Confidential} > \text{Unclassified}$$

With this setup we can imagine user $a$ classified as *Confidential*, and user $b$ classified as *Top Secret*. User $a$ may access objects with their security level or below, in this setup being objects with the classification Confidential or Unclassified. User $b$ on the other hand is classified with the highest security level (Top Secret) and is therefore able to access objects of all classifications.

As they elaborate, this concept of only being able to view objects with the same or a lower security level is called *No-read-up*. Additionally they present another analogue concept named *No-write-down*. It states that subjects shall only be able to modify objects with the same or a higher security level. The combination of these two concept, in comparison to DAC restricts the flow of information. Users who are able to view information of a security level are not able to write it to objects of a lesser security level, mitigating leaking of more secure information to weaker classified subjects.

Additionally Mandatory Access Control (MAC) also offers a possible solution for the second problem previously mentioned for DAC. While in DAC a user and a subject are the same concept, this is not the case in MAC, as mentioned by Samarati and de Vimercati (2001, p. 148). This split allows a process, running on behalf of a user, to not implicitly be run with the full access of the user. Instead it might also be run with less permissions, aiding in the avoidance of running possibly malicious actions, trying to exploit the user's elevated access.

Concluding, MAC focuses on strict security, alleviating risks that may be present in discretionary policies, but simultaneously making it a less flexible kind of policy.

### 2.3.3 Role-based Access Control

Although from some perspective RBAC may be categorized as just being a non-discretionary policy, as stated by Hu et al. (2006, p. 7), it most often is seen as the third big category next to DAC and MAC. Hu et al. (2006, p. 7) explain the basic idea of RBAC as adding the concepts of roles and operations to the already known abstractions subjects and objects. In the previously introduced policies

subjects are directly authorized to perform certain actions on some resources. With the addition of roles, which may be described as a named set of permissions or operations on resources, an additional form of indirection is added. Now a subject is assigned to a role, while the role defines the permissions on resources the subject has. This is visualized in Figure 2.2.

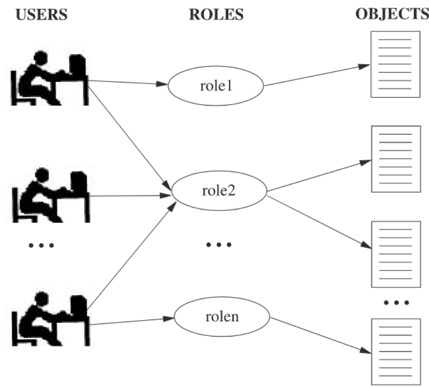



**Figure 2.2:** Indirection of access through roles (Samarati & de Vimercati, 2001, p. 182)

A big advantage is that this authorization policy represents the structure of an organization much more closely than other alternatives, making it easier implement the actual roles of employees in an organization into access control. As mentioned by Ferraiolo et al. (1995, p. 241), it also captures the fact that resources of an organization usually are not owned by an individual, in comparison to DAC. An additional advantage Hu et al. (2006, p. 7) display is an ease in authorization management. Managing the access of all individual users in the setting of an organization is tedious work, and with a certain amount of users also becomes unclear and generally hard to manage. With the indirection of roles on the other hand the authorization management becomes noticeably easier, since the process is split into two parts. First general roles with their respective permissions are defined, while in the second step users just need to be assigned to said roles. On the occassion that a user shall later get or lose certain permissions the process is simplified to assigning or unassigning a role. If the permissions of users need to be adopted, only the role needs to be modified, not the access of each individual user. So it generally can be said that RBAC is a well suited choice as access control policy for applications used by typical organization structures.

Through the flexibility of Role-based policies, it is possible to modify the above described base concept in various ways. Some works, such as Samarati and de Vimercati (2001, p. 181-183) and Ferraiolo et al. (1995) describe RBAC as one role-based model with numerous configurations, other works, such as Hu et al. (2006, p. 16-18) use a deeper separation and split the approach into different mod-

els of RBAC. Different possibilites to configure role-based models are presented in the following.

**Role Scopes** What a role describes exactly is not fixed and allows some freedom. As explained by Samarati and de Vimercati (2001, p. 181) a role can describe a whole set of responsibilities, reflecting the actual role of a person in a company. This might for example be ”manager” or ”employee”, while users would usually be assigned a single or possibly a few roles. On the other hand, roles may also be created to represent individual tasks like ”transfer_money” or ”manage_account” in an exemplary setting of a bank. In such a setup users would usually not only have one role, but all roles representing their responsibilities.

**Role hierarchies** Organization structures are usually in the form of a hierarchy. People with less responsibilities might be imagined at the bottom of the hierarchy, people with lots of responsibilities, like managers, at the top. One important aspect of this structure is that the responsibilities which the different eployees have are not necessarily exclusive. As Ferraiolo et al. (1995, p. 244-245) note, in a hierarchy employees higher up usually also have all responsibilities of employees below them. An example they mention (visualized in Figure 2.3), in the context of a hospital, is that a doctor has unique responsibilities but also all priviledges of an intern. This natural hierarchy can very well be translated into

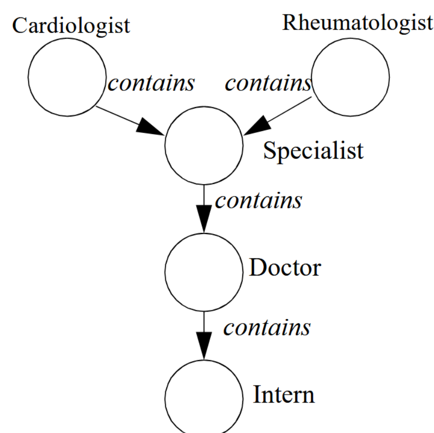



**Figure 2.3:** Hospital example of role hierarchy (Ferraiolo et al., 1995, p. 244)

a role hierarchy in RBAC, where roles higher (”doctor” in the example) in the hierarchy contain roles which are below them (”intern” in the example). As a consequence the permission and objects less powerful roles have access to, do not need to be redundantly redefined in more powerful roles. Instead the roles higher in the hierarchy inherit these, and only new permissions and objects which are accessible to the role need to be defined. This greatly decreases the authorization

management work, not only while creating new roles but also while maintaining them.

**Separation of Duties** Role exlcusivity is another aspect that might, but does not have to be defined in a policy, as elaborated by Samarati and de Vimercati (2001, p. 181 - 183). There might be instances in which a user shall not be able to have certain roles at the same time, while in other use cases acting with multiple roles is not problematic or even intended. One example they provide where exclusivity of roles might be needed is to enforce static *Separation of Duties*. Some tasks might be so impactful that no single user shall be able to perform then on their own. Instead at least two users shall be needed to perform said task cooperative. Exclusivity of roles is one way to enforce this by having two roles, each being able to perform one part of one bigger task, with both roles being mutually exclusive, meaning no user can have both roles at the same time. For completion, it should be noted that Separation of Duties may also be implemented in a dynamic fashion, where exclusivity checks could for example be done during runtime, when the resource is accessed, as noted by Samarati and de Vimercati (2001).

The above is suppossed give an understanding of the background for this thesis and is not exhaustive. While there are various additional aspects to modify Role-based Access Control to suit individual cases, for the sake of simplicity they are not elaborated on.

## 2.4 Microservice Database Consistency

One common problem in a microservice architecture is keeping databases consistent, as explained by Aydin and Çebi (2022). The encapsulation of multiple calls to the database in form of a transaction is manageable when working with a single service. But with the introduction of multiple (micro-)services, individually persiting data on one or more databases, transactions spanning multiple services may be required. They elaborate that this is a common problem in a microservice architecture using the Database per Service pattern. Their proposed solution is to implement the Saga pattern to elevate intra-service transactions to inter-service transactions. The base concept consists of every participant in a inter-service transaction to offer some local transaction $t_x$ and the inverse transaction $t_{-x}$, to roll back the last made changes, if necessary. By chaining the local transactions of multiple services, a single in inter-service transaction is formed. To achieve consistency, if some $t_x$ of service $x$ fails, all previously executed $t_n$ are rolled back via $t_{-(n-1)}$ and the initial state before the inter-service transaction is attained.

As elaborated by Aydin and Çebi (2022), there are two common variations of

the saga pattern. In an orchestration-based saga (see Figure 2.4) one central orchestrator is responsible for communicating which service is to execute which local transaction in which order. In a choreography-based saga (see Figure 2.5) no such orchestrator exists, and the services themselves communcate which each other to execute their local transactions in the correct order.

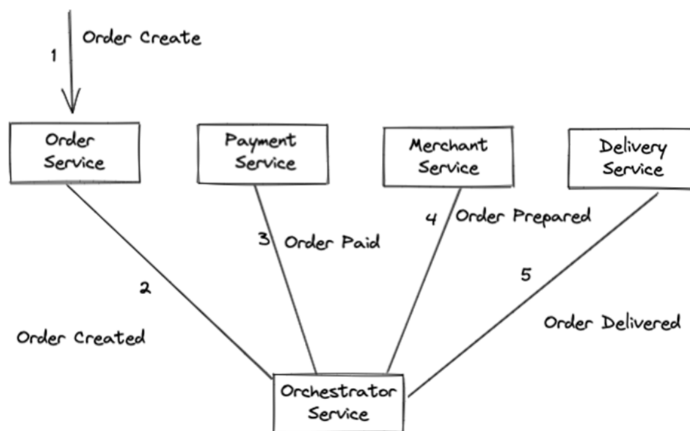



**Figure 2.4:** Example of orchestrator-based saga (Aydin & Çebi, 2022)

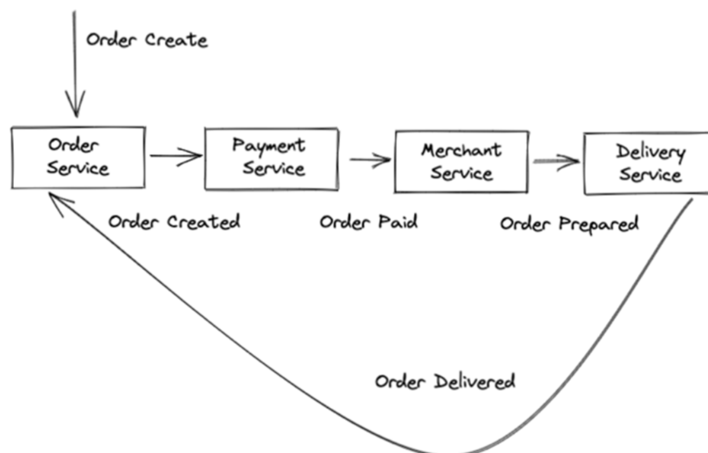



**Figure 2.5:** Example of choreography-based saga (Aydin & Çebi, 2022)

# 3 State of Technology

With a base understanding of the theory of access control, the next step is to add such to an application. While it is a possibility to design and implement a custom solution, there are already developed open-source services and libraries that may be integrated instead. The big advantage of a custom solution is that it is solving the exact problems that the application has. The noticeable disadvantage is that such a custom implementation is not as trivial as might be expected, as noted by Samarati and de Vimercati (2001, p. 138). Using open-source solutions alleviates that complication, while adding the price of not fitting as well as a custom solution.

This chapter describes currently available open-source implementations of access control and their modeling languages, which is of interest for two main reasons. First, since multiple developers work on SCA Tool, the complexity of a language should not be unnecessarily high, but still be expressive enough to be easily understandable. Second, and most importantly, the capabilities of a modeling language correspond to the internal structure of the access control system. Consequently it is an adequate representation of what the respective system is able to represent and its limitations.

It should be noted, that the following list is not exhaustive, but a selected subset, built during the research for this thesis. It is restricted to viable options for the current implementation of SCA Tool.

## 3.1 Ory Keto

Since SCA Tool is currently using Ory Kratos for authentication, Ory's access control tool Keto was a candidate of interest. As stated by GitHub (2025), Keto is an open-source alternative to Google's authorization system Zanzibar. Like Zanzibar it is build upon Access Control List (ACL)s, but also supports policies such as RBAC and Attribute-based Access Control (ABAC)(Ory, 2025a). It is written in Go but they provide an Software Development Kit (SDK) for various languages, so Keto may also well be used in applications not written in Go (Ory,

n.d.). At the time of writing this thesis it is marked as being in the *sandbox* stage, which Ory (2025c) describe as the earliest, experimental stage, with the possibility of major API changes.

As stated by Ory (2025a) and GitHub (2025), additionally to Ory Keto, Ory also offers their *Ory Permissions* system via *Ory Network*. It should be noted that this generally is a paid service, compared to self-hosting Keto.

To easily create access control models, Ory offers their Ory Permission Language (OPL). Ory (2025a) describe OPL's syntax as a subset of TypeScript, making it a familiar language for most developers. A simple example, provided by Ory (2025a), is presented in Listing 3.1, representing a simple authorization model in OPL.

```
import { Namespace, Context } from "@ory/keto-namespace-types"

class User implements Namespace {}

class Document implements Namespace {
    // All relationships for a single document.
    related: {
        editors: User[]
        viewers: User[]
    }

    // The permissions derived from the relationships and context.
    permits = {
        // A permission is a function that takes the context and returns
        // a boolean. It can reference `this.related` and `this.permits`.
        write: (ctx: Context): boolean =>
            this.related.editors.includes(ctx.subject),
        read: (ctx: Context): boolean =>
            this.permits.write(ctx) ||
            this.related.viewers.includes(ctx.subject),
    }
}
```

**Listing 3.1:** Simple OPL example (Ory, 2025a)

When interpreting the example according to the OPL specification provided by Ory (2025b), it shows a model with two types of entities, namely `User` and `Document`. `User` presumably models a type of subject, while `Document` represents a type of object. Two kinds of relations are defined to possibly exist between a document and a user. Multiple users may have the relation `editors` and/or the relation `viewers`. Additionally, permissions are defined, namely a `read` and a `write` permission. By analyzing the specification, the following may be inferred about the defined permission assignment:

**”write”-permission** The `write` permission is given to users with the relation `editors` to the document.

**”read”-permission** The `read` permission is given to a user if they either have the `write` permission or if they have a `viewers` relation to the document. The inheritance of permissions is of special interest here: Every user with `write`-permissions implicitly also has `read`-permissions. Since a user, writing to a document also needs to view (or read) the document, this permission-inheritance allows to define exactly that constraint. Consequently it is not necessary to explicitly give users with the `editors` relation the `read` permission. This is especially beneficial for larger models with more types of relations and more defined permissions.

The documentation about Ory Keto, provided by Ory (2025a), is rather limited. It does give introductions into some aspects as subjects, object, relationships between them and their API, but compared to other Keto alternatives lacks deeper information into the tool and its usage.

## 3.2 Casbin

Casbin is an open-source library offering the addition of access control to applications. It offers various kinds of policies, including but not limited to types of ACL, RBAC and ABAC (Casbin, n.d.-d). Casbin (n.d.-b) mark Casbin as ”production-ready” for eight languages, with a language-specific library for each:

- Go (Casbin)
- Java (jCasbin)
- Node.js (node-Casbin)
- PHP (PHP-Casbin)
- Python (PyCasbin)
- .NET (Casbin.NET)
- C++ (Casbin-CPP)
- Rust (Casbin-RS)

Casbin (n.d.-a) and Casbin (n.d.-e) explain the concept of how models can be built with Casbin. To adapt the model to different kinds of policies, the base concept is modified or sections added to it. Since the kind of modeling Casbin uses might not be as straight forward as alternatives, two examples, provided by GitHub (n.d.-b) are used to explain it. Listing 3.2 shows a minimal ACL model, while Listing 3.3 shows a slightly more advanced model, representing RBAC.

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = r.sub == p.sub && r.obj == p.obj && r.act == p.act
```

**Listing 3.2:** Basic Casbin ACL model definition (GitHub, n.d.-a)

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

# Adding roles to model
[role_definition]
g = _, _

[policy_effect]
e = some(where (p.eft == allow))

# Adapting matcher to not only match subject, but also roles of subject
[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

**Listing 3.3:** Simple Casbin RBAC model definition (GitHub, n.d.-c)

**Request Definition** With Casbin it is possible to define what a request looks like. As explained by Casbin (n.d.-a) and Casbin (n.d.-e) it is possible to define the argument name and what arguments are needed for a request. The display case above shows the regular format of the subject (`sub`) for which the access is requested, the object (`obj`) on which said access is requested and the type of access, or action (`act`) which is requested.

**Policy Definition** Next to a model configuration (Listing 3.2 and Listing 3.3), as explained by Casbin (n.d.-e), the actual policy rules are defined. Note that these rules are not part of the model configuration and therefore are not shown in the listings. These rules represent which subject has which permissions or may perform which actions on what objects. The policy definition in a model configuration, as shown in the example listing, defines the format for said rules.

**Matcher** Casbin (n.d.-e) and Casbin (n.d.-a) explain that a matcher may be seen as a function, getting a request and a policy rule as input, evaluating said input according to its configuration, and returning the policy result (`allow` by default) for matching rules. Consequently for a request this evaluation needs to be done for every policy rule, storing each result in `p.eft`. The matcher by itself does only have an indirect impact on the evaluation of a request, but is the basis for the later explained *Policy Effect*. The matcher, shown in Listing 3.2, likely is the most trivial implementation. Casbin (n.d.-a) elaborate that it simply compares the subject, object and action of a request with the same components of a policy rule. As previously explained, for matching policy rules, the matcher then returns the policy result for said rule.

**Role-based Matcher** To modify the simple ACL model example (Listing 3.2) to represent a RBAC model (Listing 3.3), only a few changes are necessary. Namely a new section for roles is added and the matcher needs to be adjusted to include this new addition in its matching. Casbin (n.d.-c) show that the newly added section simply adds the capability of inheritance. Since there is no explicit distinction between a user and a role, the only necessary addition to create a RBAC model is the possibility to allow users to inherit permissions from a role. Additionally the matcher is adjusted to compensated for this new concept of inheritance. It now can no longer simply compare the subject of the request with the one found in policy rules (previously done as `r.sub == p.sub`), but needs to also check if a role, the requested subject inherits from, matches. With these two modifications a simple RBAC may be created.

**Policy Effect** Casbin (n.d.-a) explain that with the matcher providing its result on a per-policy-rule basis, the policy effect now models how these individual results are combined into the single final result of the request. The most common version may be seen both in Listing 3.2 and 3.3. It may be translated to: If the matcher found at least one matching policy rule, with a policy result of `allow` the request is allowed.

The above are only basic examples and Casbin does support more kinds of policies than just ACL and RBAC. Casbin offers the advantage of a highly flexible modeling language. The modification of entire inner working of models is possible by adjusting the previously explained sections of a model configuration. This in itself is positive, but looking at it from a contrary perspective, Casbin's flexible approach also does have downsides. To choose a suitable configuration for an application, the model needs to be understood. The complexity of the language does make this step noticeably more difficult than alternatives, as also mentioned by Ory (2025a).

An additinal downside, mentioned by Casbin (n.d.-c), is that everything is represented as strings without any typing. This could lead to easier misconfigurations, since no type checking can be done. One example they mention in the context of RBAC is the fact that users and roles are not distinguished by Casbin. So while a flexible language does bring the likely imagined upsides, at least in this case more caution is necessary as well.

## 3.3 Open Policy Agent

As stated by Agent (n.d.-c), Open Policy Agent (OPA) is an open-source service usable to introduce access control into applications. However, Styra, the current maintainer of OPA also offers enterprise variants as *Enterprise OPA* and *Declarative Authorization Service (DAS)* (Agent, n.d.-d)(Styra, n.d.). Compared to other alternatives specialized to only implement access control, Agent (n.d.-c) explain that it is kept more universal, and may be used to implement various kinds of policy enforcement. Usecases they mention include examples such as access control, cluster workload-balancing and download source-restriction. So, OPA very much may be qualified as a highly versatile tool. For the scope of this thesis though, the aspects of interest are the access control capabilities of OPA. Agent (n.d.-a) state that OPA supports RBAC and ABAC policies, although the language seems flexible enough to possibly also support other kinds. It offers integrations into various programming languages and frameworks, according to Agent (n.d.-b) namely in the scope of JavaScript/TypeScript, Java, C#, Go, Clojure, Rust and PHP.

Agent (n.d.-e) explain that OPA uses their language *Rego*, inspired by the language *Datalog*, for policy modeling. Compared to the modeling languages that alternatives use, which often are very much specialized for defining an access control model, Rego is a general and more versatile language. They elaborate that since its purpose is to define queries, Rego focuses on results and not control flow, which will be seen in the later example. This allows queries to be easier to read and write, and consequently maintain. While OPA is not limited to model access control, it will be the focus in the following. They present two examples for a RBAC policy model, namely a simpler version shown by Agent (n.d.-a) and a slightly advanced version shown by Agent (n.d.-f). To get a basic idea about Rego, the simple version (see Listing 3.4) is used as a visual example.

In this example, the role- and permission assignment are located in the same file as the RBAC model definition. Before the actual query definition, the default policy result is set. In this case, as in most cases, it is set to `false`, meaning if the user does not have the requested permission, the request is denied. Afterwards, the actual permission check is performed, evaluating if the user has a role with the requested permission on the requested object.

```
package rbac.authz

# user-role assignments
user_roles := {
    "alice": ["engineering", "webdev"],
    "bob": ["hr"],
}

# role-permissions assignments
role_permissions := {
    "engineering": [{"action": "read",  "object": "server123"}],
    "webdev":      [{"action": "read",  "object": "server123"},
                    {"action": "write", "object": "server123"}],
    "hr":          [{"action": "read",  "object": "database456"}],
}

# logic that implements RBAC.
default allow := false
allow if {
    # lookup the list of roles for the user
    roles := user_roles[input.user]
    # for each role in that list
    r := roles[_]
    # lookup the permissions list for role r
    permissions := role_permissions[r]
    # for each permission
    p := permissions[_]
    # check if the permission granted to r matches the user's request
    p == {"action": input.action, "object": input.object}
}
```

**Listing 3.4:** Rego RBAC example (Agent, n.d.-a)

Concluding, OPA offers noticeable flexbility to implement access control in a variety of languages. One downside such flexibility often brings is lack of restriction enforcement. In the context of access control this may be offered distinction between different entity types, as for example subjects, roles and objects. While less restrictions do not impair the functionality in any way, configuration errors may be more difficult to catch and could lead to less robustness of the policy model.

## 3.4 OpenFGA

As OpenFGA (n.d.-c) elaborate, OpenFGA is an open-source access control service, owned by Cloud Native Computing Foundation (CNCF). As with many others it is inspired by Google's authorization system Zanzibar. They note that OpenFGA is capable of working with RBAC, ABAC and Relationship-based Access Control (ReBAC) policies. While the OpenFGA service is written in Go,

they do offer SDKs for various programming languages, making OpenFGA usable for a range of applications. They claim to support Java, .NET, JavaScript, Go and Python

As explained by OpenFGA (n.d.-a), OpenFGA offers their own language to configure access control models. While JavaScript Object Notation (JSON) is technically used internally to communicate model information, they do offer a Domain-specific language (DSL) for easier configuration and readability. As previously done, the capabilities of their DSL are presented along an example (see Listing 3.5), provided by OpenFGA (n.d.-a).

```
model
schema 1.1

type user

type domain
relations
    define member: [user]

type folder
relations
    define can_share: writer
    define owner: [user, domain#member] or owner from parent_folder
    define parent_folder: [folder]
    define viewer: [user, domain#member] or writer or viewer
                                            from parent_folder
    define writer: [user, domain#member] or owner or writer
                                            from parent_folder

type document
relations
    define can_share: writer
    define owner: [user, domain#member] or owner from parent_folder
    define parent_folder: [folder]
    define viewer: [user, domain#member] or writer or viewer
                                            from parent_folder
    define writer: [user, domain#member] or owner or writer
                                            from parent_folder
```

**Listing 3.5:** Simple OpenFGA example (OpenFGA, n.d.-a)

**Object Type Definition** As OpenFGA (n.d.-a) explain, object types are defined, modeling the different kind of subjects and objects that play a role in the system. In Listing 3.5 these are `user`, `domain`, `folder` and `document`.

**Relations** As also done in Ory Keto's OPL (Section 3.1) and Permify's modeling language (Section 3.5) relations are defined, further elaborated by OpenFGA

(n.d.-a). Semantically, a relation can represent three differentiable kinds of relationship. Trivially it can define a relation between two objects. One example is the `parent_folder` relation (see Listing 3.5), describing a hierarchy, where in this case a document or folder might be a child node of another folder. A relation might also define the role of a subject to an object. This can be seen with relations like `owner` and `viewer` in the presented example. Note that, a user is treated as a logical subject here, eventhough it is technically modelled as an object. The third kind of relation can best be described as a permission. An example in Listing 3.5 is the `can_share` relation. It describes a single action, which later might be queried to determine if a subject has that relation to the object, or rather has that permission. So, while these different kinds of relations are syntactically equal, they do describe different aspects.

**Type Restrictions** To further limit possible relationships between objects, OpenFGA (n.d.-a) explain that it is possible to add restrictions to the type of related object. These kind of restrictions can for example be seen in Listing 3.5 as listed types or relations in brackets, after the definition of a relation. While it does not seem productive to iterate through the different kinds of restrictions, they can be summarized as being able to restrict what kind of object may have the defined relation with the relation-owning object.

**Relation Referencing** Another highly useful feature OpenFGA's modeling language offers, is the possiblity to reference relations in other relations, a similar concept as inheritance in RBAC. Previously it was shown that it is possible to directly assign an object (most often semantically a subject) or a group of such as having some relation. OpenFGA (n.d.-a) furthermore explain that it is also possible to assign a relation just by the subject already having another relation. Examples of this may be seen in Listing 3.5 as for example every subject with the `owner` relation on a folder automatically also has the `viewer` relation to it. Additionally, referencing may also be done with more than one indirection. The second portion of the same example (the `viewer` relation of a folder) states that every `viewer` of the parent folder automatically also has the `viewer` relation to a child folder. So in this case a relation of a relation was used to potentially assign yet another relation. Generally it may be said that for models where certain object-relationships are known during the configuration process, referencing can very much ease the process and later maintaining.

This example does not exhaustivly describe the possibilities of OpenFGA and their modeling language, but is suppossed to show the concept they use and consequently give an insight in the flexbility and restrictions they put on such models. One example of interest for the thesis is the concept of *Custom Roles* in the context of RBAC. OpenFGA (n.d.-b) describe the possibility of allowing the creation of roles at runtime. This could either be by the application or, if

delegated, by the user of the application. With this, users may be given the opportunity to customize an authorization system, for example in the context of their organization. Consequently, such a feature might be of high interest for certain applications.

## 3.5 Permify

Permify (n.d.-c) state that Permify is an open-source authorization service, similar to Ory Keto (Section 3.1) and OpenFGA (Section 3.4), being inspired by Google's authorization system Zanzibar. As OpenFGA, Permify supports configurations to model RBAC, ReBAC and ABAC policies (Permify, n.d.-d). Similar to alternatives, next to the service they offer SDKs for various languages. According to Permify (n.d.-b) namely JavaScript/TypeScript, Java, Node.js, Go and Python.

Analog to alternatives, Permify offer their own configuration language to create so called schemas, as described by Permify (n.d.-d). Structurally it has noticeable similarities to Ory's OPL and especially to OpenFGA's configuration language. Since not only the modeling language, but also the corresponding features of OpenFGA and Permify have such a high correlation, the following explanation is kept brief, to reduce redundancy. Best efforts were made to recreate the example of OpenFGA's modeling language (see Listing 3.5), according to Permify (n.d.-d). Said example of Permify's modeling language is visualized in Listing 3.6.

As can be seen, a similar approach of defining objects (here called entities), and adding assignable relations to them is used. Restricting entities of certain types and referencing other relations are concepts that are found in Permify's modeling language as well. What Permify does allow in comparison to others, is a syntactical separation between different kinds of relations into regular relations and actions.

```
entity user {}

entity domain {
    relation member @user

}

entity folder {
    relation parent_folder @folder

    relation owner @user @domain#member
    relation writer @user @domain#member
    relation viewer @user @domain#member

    action own = owner or parent_folder.owner
    action write = own or writer or parent_folder.writer
    action view = write or viewer or parent_folder.viewer
    action can_share = write
}

entity document {
    relation parent_folder @folder

    relation owner @user @domain#member
    relation writer @user @domain#member
    relation viewer @user @domain#member

    action own = owner or parent_folder.owner
    action write = own or writer or parent_folder.writer
    action view = write or viewer or parent_folder.viewer
    action can_share = write
}
```

**Listing 3.6:** Recreation of OpenFGA example by OpenFGA (n.d.-a) in Permify's modeling language

# 4 Requirements

The below defined requirements formally describe what the implemented solution shall achieve. They are split into functional and non-functional requirements and follow the standard defined by SOPHIST GmbH (2024). As described by SOPHIST GmbH (2024) functional requirements describe an expected behaviour regarding the result, while non-functional requirements specify technical characteristics of some behaviour.

## 4.1 Functional Requirements

The functional requirements describe the features that the solution shall add to SCA Tool.

### 4.1.1 Users

**F-1:** SCA Tool shall provide users with the ability to have private projects (meaning only they have access to it).

### 4.1.2 Addition of Organizations

**F-2:** SCA Tool shall provide users with the ability to be part of organizations, giving members shared access to resources.

**F-3:** SCA Tool shall provide users with the ability to create organizations.

**F-4:** SCA Tool shall provide users with the ability to invite users to join an organization they are part of.

**F-5:** SCA Tool shall provide users with the ability to view and manage members of an organization they are part of.

**F-6:** SCA Tool shall provide users with the ability to edit an organization they are part of.

### 4.1.3 System Administration

**F-7:** SCA Tool shall provide system administrators the ability to shutdown and restart the system.

**F-8:** SCA Tool shall provide system administrators the ability to view a list of all users.

**F-9:** SCA Tool shall provide system administrators the ability to view the profile of a user.

**F-10:** SCA Tool shall provide system administrators the ability to enable and disable user login and -registration.

**F-11:** SCA Tool shall provide system administrators the ability to log users out.

**F-12:** SCA Tool shall provide system administrators the ability to suspend and remove users.

**F-13:** SCA Tool shall provide system administrators the ability to view and edit all organizations.

**F-14:** SCA Tool shall provide system administrators the ability to remove organizations.

**F-15:** SCA Tool shall provide system administrators the ability to monitor system resources.

**F-16:** SCA Tool shall provide system administrators the ability to view logs of the SCA Tool backend.

### 4.1.4 Access Control

**F-17:** SCA Tool shall limit access to resources based on permissions.

**F-18:** If a user does not have sufficient permissions to execute an action, SCA Tool shall not show related UI-elements.

**F-19:** SCA Tool shall provide users the ability to create, modify and remove roles, grouping permissions on a type of resource.

**F-20:** SCA Tool shall provide organizations sensible default roles and a default assignment.

## 4.2 Non-Functional Requirements

The defined non-functional requirements describe how the implementation shall behave.

### 4.2.1 Performance

**NF-1:** Permissions-checks shall add a maximum latency of 15ms.

**NF-2:** Permissions-check results in the backend shall be cached.

**NF-3:** Available permissions shall be cached by the frontend.

### 4.2.2 Portability

**NF-4:** The access control system shall be deployable via Kubernetes.

**NF-5:** The access control system shall be deployable for local development.

### 4.2.3 Maintainability

**NF-6:** The added components shall fit into SCA Tool's architecture.

**NF-7:** Permission-checks shall consist of a single function call.

**NF-8:** Modification of the access control configuration shall be simple.

### 4.2.4 Reliability

**NF-9:** All external access to SCA Tool needs to pass an authorization-check.

**NF-10:** SCA Tool shall never give unauthorized users access in case of a system error.

### 4.2.5 Integrity

**NF-11:** Transactions that span multiple services shall never lead to inconsistencies in the databases.

# 5 Considerations

Previous chapters present various aspects on how access control can be integrated into SCA Tool. Different kinds of policies are available (see Chapter 2) and tools offer their pre-made solutions (see 3). This chapter discusses the different options, weights them and explains why a variation of RBAC was chosen for SCA Tool. It additionally present why customization of roles is relevant to fit different kinds of customers, and finally why the pre-made solution Permify was chosen to be integrated.

## 5.1 Best Fitting Policy

The three kinds of policies, presented in Chapter 2.3, are possible candidates for SCA Tool. Every kind offers individual advantages, which are weight in the following.

**Mandatory Access Control** Mandatory policies are immensely strict and most suitable for applications where security is of utmost importance. Simoultanesouly, it lacks in flexibility compared to other policies. SCA Tool is not dealing with highly secure information and the absolute prevention of data leakage inside an organization is not the highest priority. Consequently, MAC is considered too restrictive and not well suited for the application.

**Discretionary Access Control** Discretionary policies on the other hand prioritizes flexibility, allowing fine-grained permissions to be assigned to users. It allows for various administrative policies to be used, making it suitable for a variety of applications. A hierarchical adminstative policy is of interest for SCA Tool, having central administrators but also the possibility of delegating such priviledges. Central administration is needed on two different layers, the system and organizations. The system itself needs administation to monitor and manage users and organizations. Organizations need the possibility to manage themselves, including members, authorization and in the near future, billing. While central authorization is sufficient for system-management, management

inside organization would benefit from hierarchical administation. Organization hierarchies could be modeled more precisely, for example having one organization administrator but also multiple managers, with less managing-priviledges or only a subset of resources as responsibility.

**Role-based Access Control** Role-based policies also offer aspects that work well with applications like SCA Tool. The concept of roles is suitable to model organizations, not only to represent their internal hierarchy but also by grouping permissions into responsibilities. Another advantage is the ease in administration work it offers. Instead of assigning users individual permissions on resources, the process is simplified by grouping permissions and/or objects. Consequently, a role-based approach is chosen for the application. For the administation of organizations, discretionary hierarchical administation is determined to be most suitable for SCA Tool. It allows for central authorization of an organization owner, simultaneously making it possible to delegate all or some administation to other members.

## 5.2 Authorization Customizability

A role-based approach offers the aspect of role customization. The simplest possibility is to only offer fixed roles to users. Thus, the roles are defined by SCA Tool, and while the user may choose the assignment, the modification of roles is not possible. While simple, it is difficult if not impossible to determine a set of roles which fit all organizations and their internal structure.

A somewhat better approach is to allow users to modify the permissions of fixed roles. While the roles are still defined by SCA Tool, slight changes can made to better fit individual users. However, larger adaptions are still not possible, and the problem of no set of roles fitting all users persists.

Consequently, the most suitable approach for SCA Tool is to allow fully customizable roles. While offering a pre-determined default set of roles, organization administrators can create new roles and set the respective permissions as needed.

## 5.3 Non Resource-bound RBAC

One consideration of a role-based approach is the downside of how *classical* RBAC (as described in most literature) implements roles. Traditionally roles capture permissions on individual resources. It is important to emphasize that the resources on which the permissions are assigned, are bound to the role. For applications where resources are unique, this concept works well, but for applications where resources of one kind exist in large amounts, it is less suited.

Docker Desktop taken may be taken as an example. It can be viewed as offering fixed unique features, as managing containers, managing images and managing volumes. To now model a role that represent a *client* who is allowed to manage containers (especially starting and stopping them), and a *maintainer* who is allowed to manage images and volumes, this approach fits well. This is because the features are presented as unique objects, only existing once in the application. In such a case it makes sense to directly bind the resources to the roles.

On the other hand, for applications which work with large amounts of individual objects of the same kind, this approach introduces an issue. Reusing the example of Docker Desktop, it may be necessary to not only model the individual features as objects, but also fine-grained resources, meaning the containers, images and volumes themselves. This could for example be required to limit the access of subjects to only a subset of these resources. Another, more relevant example is SCA Tool. In an organization, SCA Tool offers a strict hierarchy of resource types, namely Projects, Code Units and Code Unit Versions (or Versioned Code Units). An organization can own multiple Projects, each with multiple Code Units, which again may contain multiple Code Unit Versions, in that order. With the assumption that different members of an organization work on different projects, the classical role-based approach no longer fits as well. This can be shown with a simple example.

Assuming a scenario with five projects, and members having the same responsibilities but on different projects, not one, but a maximum of five roles might be needed. Each of these roles would hold the same permissions, but since resources in classical RBAC are bound to a role, multiple role-instances might be required. In the best case scenario (displayed in Figure 5.1), all users with the same responsibilities (or permissions) on projects, need access to the same set of projects. In this case, one role is sufficient.

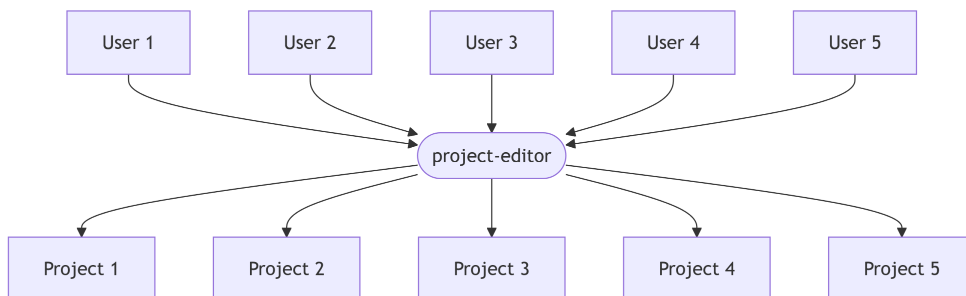



**Figure 5.1:** Classical RBAC, best-case example

A more realistic scenario is that projects are semantically split into fixed groups, meaning that there is for example one set of members that works on Project 1,

2 and 3, and another set that works on Project 4 and 5. In such a case two roles (or role-instances) would suffice, as show in Figure 5.2.

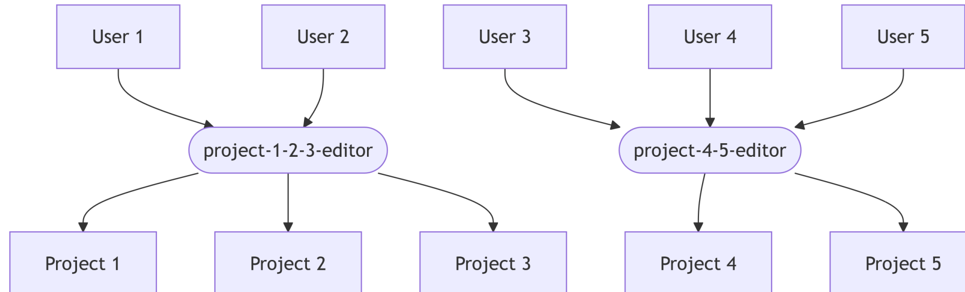



**Figure 5.2:** Classical RBAC, realistic example with grouped projects

In the worst case scenario (displayed in Figure 5.3) no fixed project-groups can be defined. The reason for this is that the smallest common denominator are single projects. In other words, no $n$ amount of projects (with $n > 1$) are only assigned as a group. In this case five roles are needed for five projects.

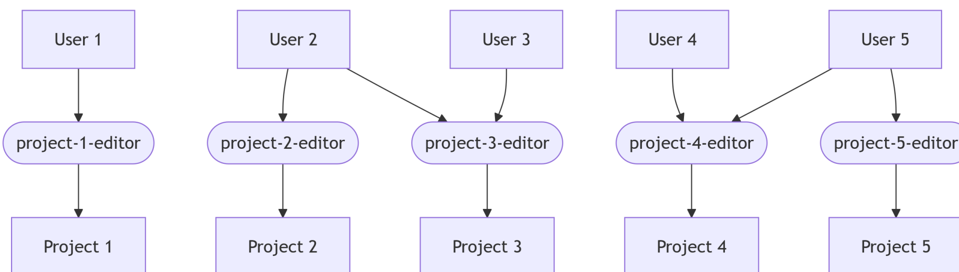



**Figure 5.3:** Classical RBAC, worst case scenario with no senseful project-groups

Such scenarios are especially frustrating, since all role instances need to be maintained individually. Usually clear naming of roles may now become confusing since roles like *project-editor* become instances like *project-1-2-3-editor* and *project-4-5-editor*. In addition, the management of permissions can no longer be done in one role, but needs to be performed in all instances.

How flexible permission-assignment needs to be for an application is dependent on it's users. If a SCA Tool organization consists of a single or couple of users, the required access control may be assumed to consist of only a few broad roles. In such a case, classical RBAC is flexible enough and the above mentioned issues would not be a concern. On the other hand, if an organization exceeds such a

small size, it may be assumed that members have different areas of responsibility, therefore working with different resources. This would lead to a higher quantity of fine-grained roles, on different subsets of resources. For such a case, the described issue of classical RBAC very much would present itself.

Since SCA Tool is meant to be accessible to all kinds of users, it is determined that the classical RBAC approach should be modified. The proposed solution is to remove the implicit binding between a role and individual resources. Consequently, a role only describes permissions, while the assignment of this permission-set to resources is separated to another step. Roles can hence be defined as:

> A role encapsulates a set of permissions on a type of resource.

While this approach alleviates the issue from resource-binding, it is important to note that it does add complexity. The process of assigning permissions on resources to a user is no longer achieved by simply assigning a role to a user. Compared to classical RBAC, it is now necessary for roles to be assigned to individual users on individual resources. This necessity for individual assignment in this approach can be seen in the re-creation of previous examples. Figure 5.4 mirrors the scenario from Figure 5.1 but for the non resource-bound approach, while Figure 5.5 mirrors Figure 5.3. Note that the assignment in Figure 5.4 is only done for two users. This is just for visual clarity and not related to the different approach.

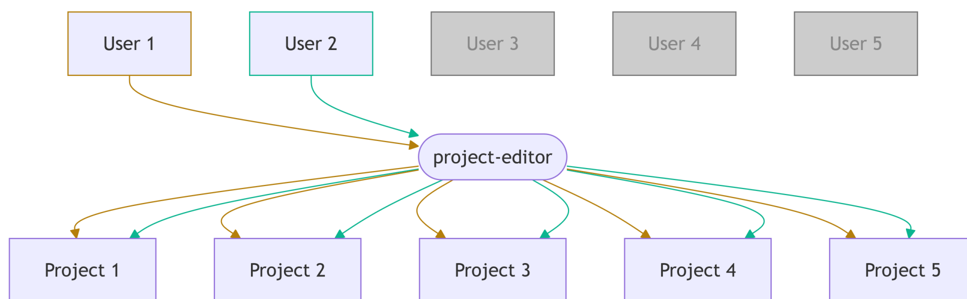



**Figure 5.4:** Non resource-bound approach of scenario in Figure 5.1. Stripped to only two users for visual clarity.

As can be seen by the increased number of arrows, the approach does increase the needed assignment work. Even though modifications, automating a noticable amount of assignments are possible, the basis of individual assignment persists. On the other hand, the Figures also show the big benefit: Only one role (according to the new definition) needs to be defined for one set of responsibilities. Consequently, role management and -maintance is clear an concise, in comparison to the redundant role instances, required in a classical RBAC approach.

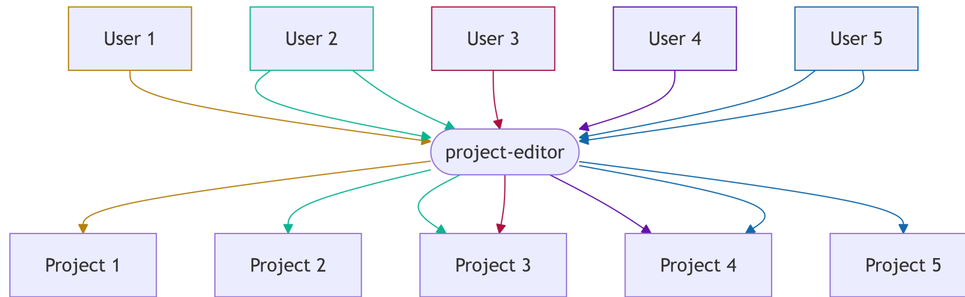



**Figure 5.5:** Non resource-bound approach of scenario in Figure 5.3

Concluding, both classical RBAC and non resource-bound RBAC have advantages and disadvantages as access control policy for SCA Tool. Classical RBAC brings more work in managing redundant roles and maintaning sensible groupings of resources. Non resource-bound RBAC on the other hand, adds more assignment-work.

For SCA Tool, the decision is made to implement a non resource-bound RBAC approach. While it introduces complexity in assignment-maintanance, its flexibility makes it a cleaner approach for both small and large organizations in SCA Tool. Additionally, a considerable amount of assignments can be automated, lessening the impact of the approaches drawback.

## 5.4 Access Control Implementation

When introducing access control into an application both a custom solution, and an available access control system are viable options. The clear advantage of using a pre-made solution is the reduced development time and the insurance of a competent implementation. Consequently, such a solution is preferred for SCA Tool, instead of a fully custom implementation.

As a variety of such solutions are potential candiates (see Chapter 3) a decision needs to be made. Multiple factors are deemed as relevant:

- **Open-Source:** The tool is required to be licensed under a permissive open-source license.
- **Java Language Support:** SCA Tool has a Spring Boot backend. The access control system needs to offer a library or SDK for Java. All systems presented in Chapter 3 support Java.
- **Role-based Access Control:** The tool needs to support RBAC.

- **Non Resource-bound Roles:** A system that (if by design or not) supports non resource-bound roles is preferred.
- **Custom Roles:** The system needs to support the creation, modification and removal of roles at runtime.
- **Type-bound Permissions:** Permissions which are bound to a resource type are preferred. This way, even permissions with the same name on different resources can be differentiated. It adds a protective layer against accidental access through bugs or misconfigurations.
- **Adequate Modeling Language:** The modeling language should be powerful enough, but not overly complicated. SCA Tool has multiple developers who might need to work with it.
- **Sufficient Documentation:** Integration of a tool with insufficient documentation can significantly increase integration-time. Unnecessary development time disfavored in a time-limited thesis.

Since Ory Keto (Section 3.1) is in an experimental stage of development and has very sparse documentation it is not preferrable. Casbin (Section 3.2) does not only not offer type-bound permissions but generally treats most values as strings. This makes giving accidental access noticeably easier. Additionally, Casbin as well as OPA (Section 3.3) use rather complex configuration languages. The hereby added complexity makes future adaption through different developers difficult. OpenFGA (Section 3.4) and Permify (Section 3.5) offer a very similar system. Both fullfill all mentioned relevant factors, disregarding the support of non resource-bound roles and classify as well fitting for SCA Tool. Since Permify did present a slightly more polished product, it is chosen as the access control system to be integrated into SCA Tool. As non resource-bound roles is not an established concept, none of the tools or their configuration languages offer a way for direct implementation. Thus, Permify by itself is not sufficient to implement the approach, and a wrapper is needed. Details on the exact structure and functionality of said wrapper are discussed in Chapter 7.

# 6 Architecture

This chapter shows the architecture of the proposed solution. Described is how the concept of organizations and authorization was introduced to SCA Tool.

## 6.1 Users and Organizations

As defined in the requirements, the concept of organizations (see Chapter 8.1.2) and personal projects (see Requirement F-1) shall be introduced to SCA Tool. Currently, organizations do technically exist in the application, however they do not actually provide the expected functionality. As visualized in Figure 6.1, every user owns an organization, which again owns the resources of the user. Every user owns exactly one organization, and every organization only has said user as a member. Consequently, the current implementation of organizations solely represents the personal projects (and other resources) of a user, with an additional level of indirection.

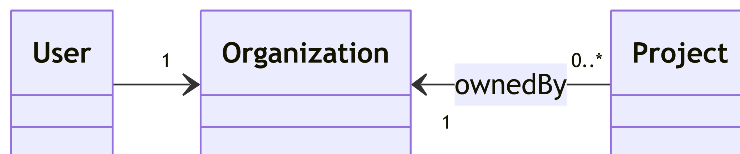



**Figure 6.1:** Current imlementation of organizations

### 6.1.1 Adding Organizations

To allow the sharing of resources through organizations the existing implementation needs to be adapted. Users need the ability to be part of multiple organizations (see Requirement F-2). And likewise, organizations need to be able to have multiple members. Thus, a membership-entity is introduced, modeling exactly that relation (see Figure 6.2).

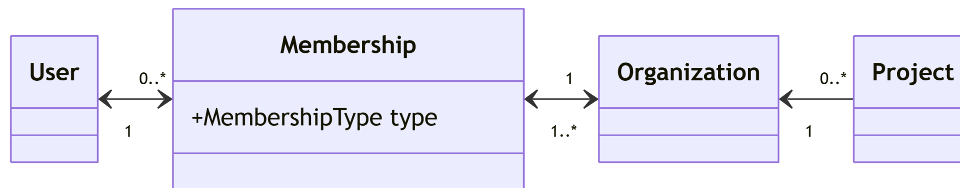



**Figure 6.2:** New imlementation of organizations

Modeling memberships as a bidirectional relation is preferred over a unidirectional relation from the organization. This way, not only the members of an organization can be easily queried, but also the organizations a user is part of. Additionally, it allows to model different kinds of relations, a user might have to an organization. Examples are an invited user, member or owner. For each organization, a user can have a maximum of one relation at any point in time. This is strictly enforced on the persistence-layer, with a membership object (see Figure 6.2) having a composite-key, built from the corresponding user and organization. To only allow one relation does not only keep the concept simple, but also has the technical advantage of no accidental multi-relations.

### 6.1.2 Personal Projects

Additionally to organizations owning resources, users shall still have the option to work on personal projects. The trivial approach would be to expand the possible ownership-relation of resources to not only be able to be owned by organizations but also by users. While representing what a user expects to see, it adds the necessity of handling two separate cases, an organization-owned resource or a user-owned resource, throughout the application.

A better manageable approach is to simply keep the current indirection via a dummy organization, seen in Figure 6.1. Thus, every user owns an additional mandatory organization, holding their personal projects. The addition of a flag in every organization-instance allows the differentiation between a personal- and actual organization, where needed, but allows for a univeral way of handling resources otherwise. Examples where such differentiation is of interest, is a different visualization in the frontend or constraints like personal organizations not allowing multiple members.

## 6.2 Authorization

This section dicusses the proposed structure of the access control integration via Permify. It is explained where in the backend authorization checks should be

done, and what potential changes need to be made to the current architecture. Additionally, to support non resource-bound RBAC to users, some translation needs to exist to produce Permify-conform authorization rules.

Currently, SCA Tool's Spring Boot backend follows the structure of an Anemic Domain Model (ADM). Cemus et al. (2016) explain the basic concept of the ADM architecture style as splitting the system into layers with different responsibilities. Data structures are then passed between the layers, each layer performing operations on or with the data. The passed data structures are anemic, meaning they are not objects with methods, but simple aggregations of data. In comparison, they describe that in a Rich Domain Model (RDM) architecture, data and operations on that data are encapsulated into objects, in the fashion of Object-oriented programming (OOP).

For the implementation of authorization in SCA Tool, a RDM architecture would have the advantage that the access-checks can be encapsulated in the methods of an object. This additionally ensures that every access-check is performed close to the resource. The further a resource is passed up from persisting storage before the necessary access-permissions are ensured, the more diffcult it gets to catch every operation that accesses the resource somewhere in its call stack. Hence, doing authorization checks as close to the resource-access as possible, as could be easily done in a RDM structure, would counteract that issue.

While a RDM architecture is assumed to work well with the integration of an authorization system, the current ADM architecture would need to be completely re-written. In the scope of this thesis, a complete refactoring of the SCA Tool backend is not practical. Since Spring Boot additionally is well suited for ADM architecture, the application's overall architecture is kept as is. While not perfect, it is the suitable approach for SCA Tool.

Consequently, for the integration of Permify into SCA Tool, two new Spring services are created (see Figure 6.3). `PermifyService` wraps the API offered

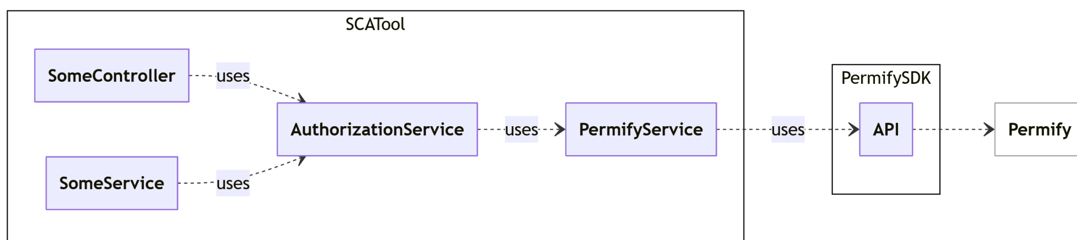



**Figure 6.3:** Integration of authorization into SCA Tool

by the Permify Java SDK and presents convenient communication-funtionality with Permify to the backend. `AuthorizationService` abstracts Permify to a

more agnostic authorization interface, usable by other services on the same layer, or controllers on a higher layer. It additionally acts as a translator between non resource-bound RBAC, presented to SCA Tool, and a Permify-conform authorization-format.

# 7 Design and Implementation

This chapter discusses how organizations, access control and the system management dashboard are realized in SCA Tool.

## 7.1 Users and Organizations

For a usable introduction of organizations the changes in architecture are not sufficient. They need to be manageable, not only by the system, but also by the members of an organization.

To introduce the necessary logic, a new component was added to the service layer. The most relevant functionalities it adds to organization-management are the following:

- Creating, editing and deleting organizations
- Managing organization-members
- Retrieving organization-user relations

### 7.1.1 Organization-Relationship Types

While the handling of organization objects themselves is rather trivial, the management of members offers interesting aspects. Next to no affiliation, users can have three types of relations to an organization: *Owner*, *member* and *invited*. Compared to members or invited users, an organization can and must have exactly one owner.

While both users with an owner- and a member-relation are part of on organization, the differentiation is important. The main reason to explicitly define one member as the owner is the added security. An owner has two priviledges that regular members lack. Compared to other members, they cannot be removed from the organization. Consequently, even if bad actors are allowed management permissions to the organizations, at least one good actor is always present. Owners', as members' access are limited according to their assigned roles, with

one exception. Even without any assigned permissions, the owner of an organization implicitly has the required acces to modify roles and role-assingments. Thus, an owner can recover their own, but also others' access under any circumstances. This is especially important since a hierarchical administration (see Chapter 2.3.1) is possible, meaning multiple members can act as role-managers.

To add users to an organizations, managing members can invite other users to join. Users cannot join an organization without being invited and organizations cannot simply add users. In other words, the decision for a user to become a member is done in cooperation between the inviting- and the invited party. Invitations are implemented as a user receiving the invited-relation to an organization, which gets promoted to a member-relation upon acceptance.

### 7.1.2 User Experience of Organization Management

Prior to the addition of organizations, the full strict hierarchical structure of resources (projects owning code-units, owning version) was correspondingly presented to the user via a tree-view in the persistent sidebar. The new resource-type of organizations is handled differently. Working on projects of different organizations is treated and visualized as a context-switch. The workspace consequetly is no longer global, but has the scope of an organization. This implementation keeps the work done in different organizations strictly separated, while not adding more complexity to the sidebar navigation.

To allow organization-managers to view and edit the organization and authorization, an organization-settings screen is added. It is accessible when working in the context of an organization, offering adminstrative functions. Members can be viewed, invited to the organization or be removed from it, revoking all access. The settings additionally allow managers to create and modify roles and update role-assignment of the members. The subset of settings which are available to a member are regulated through their permissions on the organization.

## 7.2 Access Control

This section depicts how Permify was integrated into SCA Tool to implement authorization.

### 7.2.1 Translating Non Resource-bound Roles

Permify by itself only supports classical RBAC, meaning a role captures permission on fixed resources. The previously introduced idea of non resource-bound roles on the other hand, removes that binding to resources of a role. Consequently,

roles only capture a set of permissions of a resource-type and are assigned to a user for a dynmamic set of resources of said type.

To offer the SCA Tool backend non-resource bound RBAC a translation to Permify-conforming classical RBAC is added. The concept of a role is split into role-templates and role-instances, shown in Figure 7.1. A role-template describes

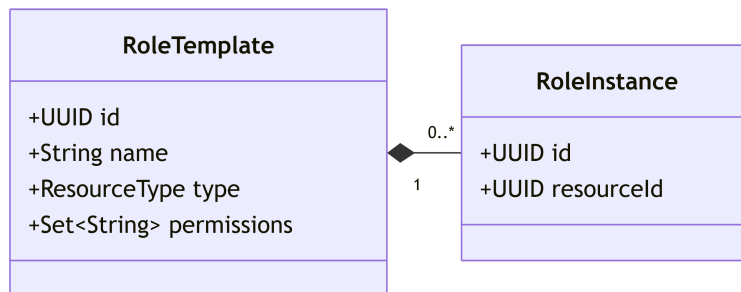



**Figure 7.1:** Relation between role-templates and role-instances

a non resource-bound role, namely a set of permissions on some resource-type. To now map such a template to a classical RBAC role, which binds a resource, a role-instance is created. It is an instance of a role-template for one resource of the specified resource-type of the template. Thus, from the perspective of classical RBAC, the combination of a role-instance with its template depicts a valid role.

The explicit separation into a template and instances does lead to redundant storing of information, regarding the authorization data persisted via Permify. This has the downside that the databases of SCA Tool and Permify need to be kept consistent, as explained in Subsection 7.2.7. While not ideal, it is required for clean role-management. To keep track of the resources for which a role-instance of a template exists and to update said instances upon changes, redundant storage is the best option.

### 7.2.2 Role Management

To provide SCA Tool with a non resource-bound RBAC system, SCA Tool's authorization-service is introduced. It offers the necessary functionalities for access-control management and acts as a translator between non resource-bound roles and classical roles, according to the concept described in Subsection 7.2.1. The most relevant functionalities it provides to the backend are the following:

- Checking if a user has some permission on a resource
- Creation, editing and deletion of roles
- Editing permissions of roles

- Editing role-assignments of a user on a resource

The scope in which a role is valid and accessible is limited to an organization, meaning roles are owned by organizations. Thus, the authorization-service offers additional functionality, with the most relevant being the following:

- Retrieving all role-assingments of an organization member
- Retrieving all role-assignments on some resource
- Retrieving the resources on which a user is assigned some role

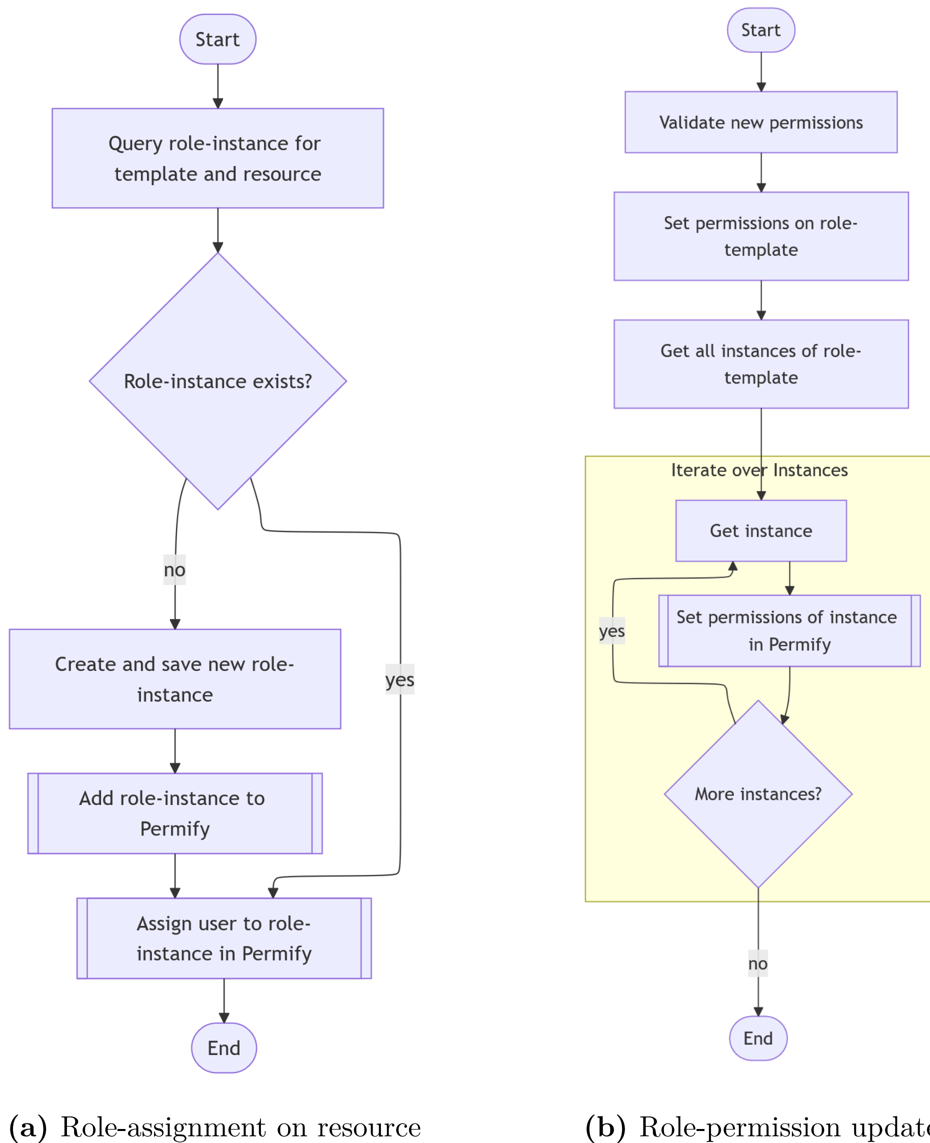
While the actual management logic of the service is rather straightforward, the required translation from non resource-bound RBAC to classical RBAC does add complexity. Changes in roles, role-assignment or permissions of a role need to be handled and traversed down to the Permify instance via the `PermifyService`. For better visualization of the necessary translation, done by the authorization-service, two examples are provided. Figure 7.2a shows the process of assigning a role on some resource to a user, Figure 7.2b of updating permissions of a role. They are representitive for most translation logic in the authorization-service, entailing updates to the role-template and the affected role-instances, and lastly the necessary calls to Permify. Figure 7.3 additionally shows the interaction between the different components for the example of assigning a role on some resource to a user (compare to 7.2a).

### 7.2.3 Handling different resources

SCA Tool handles multiple types of resources, which are relevant for an access control system. To be more precise these are organizations, projects, code-units and code-unit versions, which are all processed in vastly different fashion. This breaks down to different controllers, services and repositories for persisting storage. When compared to an access control system, each of these objects is boiled down to a resource and mostly handled in the same way.

This mismatch creates an issue, since Permify expects a resource to simply have a type and an ID, while SCA Tool offers distinct objects. To solve this mismatch, multiple options are available. Multiple wrappers for the authorization-service could be implemented, one for each resource. Another option is to only use one authorization-service and add overloading for different resources. While possible, both add code overhead, which needs to be maintained and adds unnecessary complexity.

The simplest solution is to add a new resource-interface, implemented by SCA Tool's resource classes (see Figure 7.4). It unifies SCA Tool's resources to the essential parts needed for access control, the type of resource and the id. Con-




**(a)** Role-assignment on resource **(b)** Role-permission update

**Figure 7.2:** Flowcharts of role-management examples

sequently, the authorization-service deals with universal resources, all implementing said interface.

As presented in Chapter 3.5, Permify's rules are configured through the definition of a schema. Precisely the above mentioned resources are of interest for SCA Tool. The schema additionally captures the permissions which are available on an entity, which a role might have on some type of resource. The permissions defined on an organization can be categorized into general actions on the properties of the organization, actions regarding the members and actions for role-management.

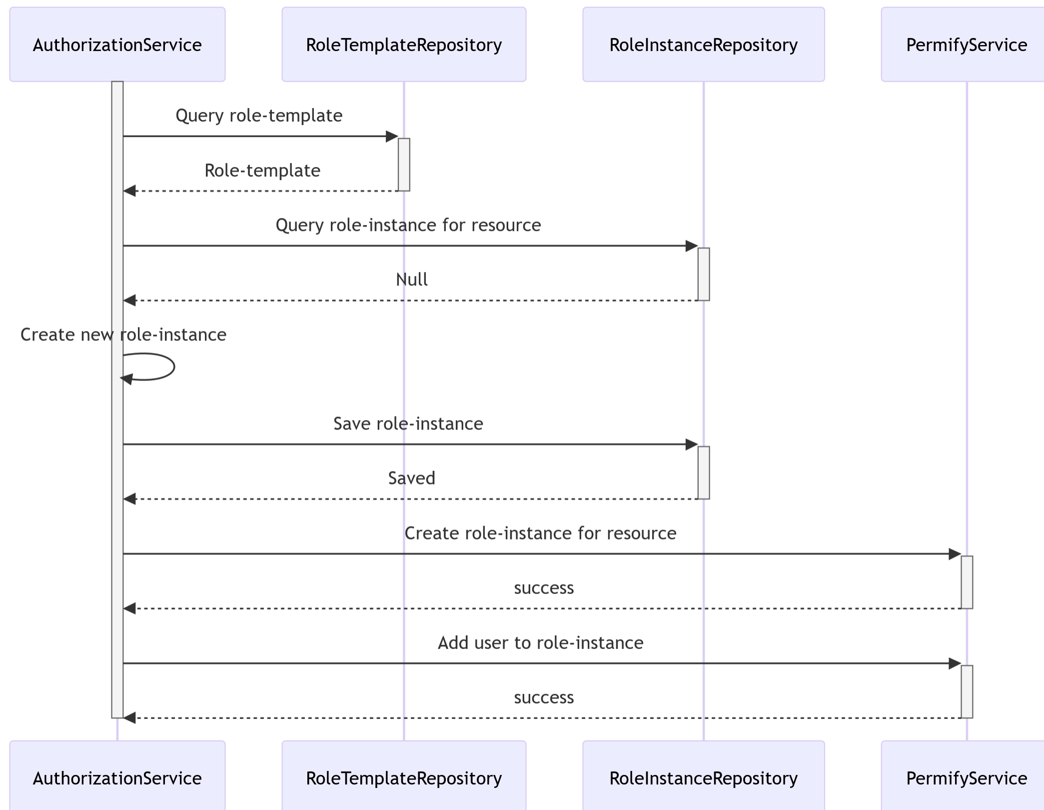



**Figure 7.3:** Interaction between SCA Tool's components during role-assignment of user on resource

They are listed in Table 7.1.

The permissions defined on projects (see Table 7.2) and code-units (see Table 7.3) mainly consist of management permissions, including creating new instances or deleting them. One outlier is the editing of governance rules, which is bound to code-units. The decision on what kind of licenses are classified as being allowed, must-ask or fully prohibited are not made on a code-unit version level, but a code-unit level. Said decision then is applied to all versions of the code-unit. Thus, the action (or permission) of editing the rules is bound to the code-unit as well.

The main features of SCA Tool, SBOM management, license governance, compliance and vulnerability management all depend on the exact components being used in software. Since different versions of software, and therefore a code-unit, may have different dependencies, these features are bound to the level of a code-unit version. The respective permissions follow accordingly and are defined on code-unit versions (see Table 7.4).

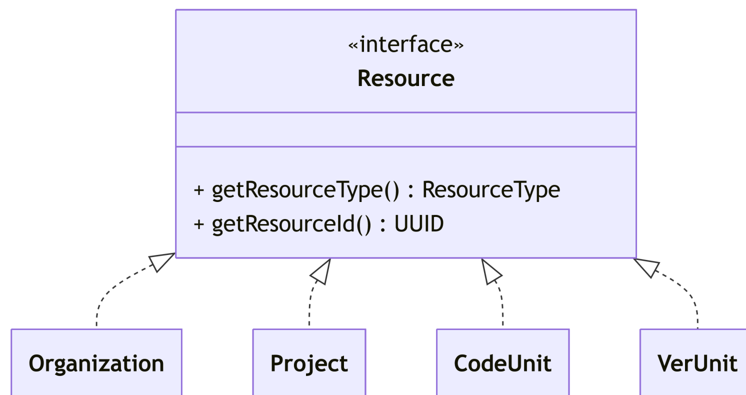



**Figure 7.4:** Resource interface, implemented by SCA Tool's resources

### 7.2.4 Authorization Enforcement

SCA Tool is built with a layered architecture, mostly being with controllers handling API requests, services implementing business logic and repositories to write and read from persisting storage. Figure 7.5 shows a simple example of a request and the involvement of the different layers.

Authorization checks can be performed on different layers, but the location should stay consistent. In the planning phase it was determined that such checks are preferably performed as close to the resource as possible. Using the repository layer is not possible, since resource-requests to it are context-agnostic, meaning it is unknown what permissions are required to access the resource. While both services and controllers are viable options, controllers are used for authorization enforcement. Although contradictory to the strategy of keeping enforcement close to the resource-fetching, further assessment during implementation lead to this decision. The functions, offered by the services, are reused for different kinds of access throughout controllers. Thus, with the current implementation of services, they can be classified as partially context-agnostic. While this can be changed, by duplicating the services' logic for different types of access, the added redundancy is not preferrable. So, the controllers are the only layer where the full context, of which permissions are required for access, is known. Consequently, access control is enforced on the controller-layer.

While authorization does restrict the resources that a user has access to, this does not implicility adapt the UI accordingly. Taking the SBOM tab of a code-unit version as an example, the intuitive consequence of a user not having permission to view the components would be to neither see the clickable "SBOM" menu-item nor the screen. However, when authorization is solely enforced in the backend,

| Organization | |
|---|---|
| General | |
| view | View organization |
| edit | Edit organization properties (e.g. its name) |
| delete | Delete organization |
| create_project | Create new projects for organization |
| Members | |
| view_members | View members of organization |
| invite_member | Invite user to organization-membership |
| remove_member | Revoke membership of user to organization |
| Role Management | |
| view_roles | View roles owned by organization |
| edit_role | Create and edit roles and their permissions |
| delete_role | Delete roles |
| view_role_assignments | View role-assignments of members |
| edit_role_assignment | Assign / Revoke member's role-assignments |

**Table 7.1:** Available permissions on organizations

| Project | |
|---|---|
| view | View project |
| edit | Edit project properties |
| delete | Delete project |
| create_code_unit | Create new code-units in project |

**Table 7.2:** Available permissions on projects

this is not the achieved behaviour. A user would still see the clickable menu-item and an empty SBOM screen. In the context of User Experience (UX), this is unintuitive and suggests false informations, as for example a code-unit version simply having no components. Hence, it needs to be possible to adapt the frontend's UI according to the permissions of a user.

Permify does offer the react library react-role[1], which tries to solve this exact issue. While it is not powerful enough to be used for SCA Tool, their enforcement features are simple but well chosen, thus being re-implemented for SCA Tool's frontend. They offer two options for frontend-code to adapt its rendering according to permissions. To for example disable components via a property

[1]https://github.com/Permify/react-role

| Code Unit | |
|---|---|
| General | |
| view | View code-unit |
| edit | Edit code-unit properties |
| delete | Delete code-unit |
| create_ver_unit | Create new versions for code-unit |
| Governance | |
| edit_governance | Edit governance rules |

**Table 7.3:** Available permissions on code-units

| Code Unit Version | |
|---|---|
| General | |
| view | View code-unit version |
| edit | Edit code-unit version properties |
| delete | Delete code-unit version |
| Content | |
| view_sbom | View components of version |
| view_governance | View governance data of version |
| view_compliance | View and download legal notices of version |
| view_security | View vulnerabilities of version |
| edit_security | Manage vulnerabilities of version |

**Table 7.4:** Available permissions on code-unit versions

field, a "can-access" boolean needs to be obtainable, with a simple function being sufficient. Additionally to completely hide components which are irrelevant for a user, like the SBOM menu-item with insufficient permissions, they provide a guard. Said guard is wrapped around the components in question and either renders them if the user has access to the components, or hides them otherwise. An example, showing both options is shown in Figure 7.1.

Since the actual authorization enforcement is solely done in the backend, communication with the frontend is necessary to transmit the user's permissions. To avoid excessive latency, the authorization data is fetched on-demand. More precisely, when the frontend needs the user's permissions on some resource, exactly these are fetched. Additionally, to reduce unnecessary requests, all fetched permissions are cached throughout the session.

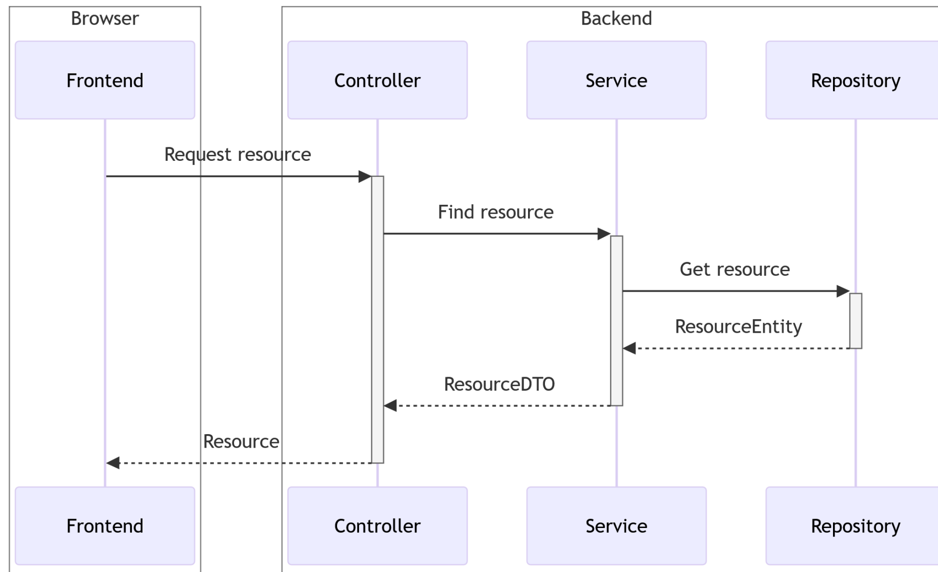



**Figure 7.5:** Resource-request being handled by the different layers of the backend

### 7.2.5 Deploying Permify

To use Permify in an application a running instance needs to be introduced into the environment. For SCA Tool two different environments exist, namely a local development environment and multiple Kubernetes deployements. For the local environment Permify offers a container image, for Kubernetes deployments Helm Charts (Permify, n.d.-a, n.d.-e). Adding the Permify docker container for local development is straightforward. An entry is added to the compose file, and a new database is added and provided to Permify.

To add Permify to the Kubernetes deployments, Helm Charts are used to generate the required files. After configuring the connection to a new database and generating the necessary secrets, Permify is deployed in the Kubernetes Clusters as well.

### 7.2.6 Making Permify Accessible to the Backend

The Permify Java SDK alleviates the necessity to build raw HTTP requests for communication between an application and the Permify instance. It wraps the API into analog nested Java objects, which can then be filled according to the API specifications, and send to the specified service location. However, building such API requests throughout the application still is not a usable way of communication. To emphasize this, Listing 7.2 shows an examplary request to check if a

```tsx
import { useUserPermissions }
    from "@/components/authorization/user-permissions-provider.tsx";
import { PermissionConditionalViewer }
    from "@/components/authorization/PermissionConditionalViewer.tsx";

export function SomeComponent() {
    const { hasPermission } = useUserPermissions();
    const project = getProject();

    return (
    <>
        {/* Always shows button, but disabled if insufficient permissions*/}
        <Button disabled={!hasPermission("PROJECT", project.id, "edit")}>
            Edit Project
        </Button>

        {/* Only renders button if user has sufficient permissions */}
        <PermissionConditionalViewer
            resourceType="PROJECT"
            resourceId={project.id}
            permission="edit"
        >
            <Button>
                Edit Project
            </Button>
        </PermissionConditionalViewer>
    </>
    );
}
```

**Listing 7.1:** Frontend adaption to render content dependent on permissions

subject has some permission on an object, build with the Permify SDK. To simplify the communication, the wrapping Spring Boot Service `PermifyService` is implemented. It offers sane functions to the `AuthorizationService` and internally builds and sends the necessary requests. Since the Permify-wrapper is only of interest to the `AuthorizationService`, it is kept hidden from other services of the backend. The most important functionalities `PermifyService` provides are the following:

- Checking if a user has some permission on a resource
- Editing users assigned to a role-instance
- Editing permissions of a role-instance

### 7.2.7 Database Consistency

For the management of SCA Tool's roles, some data needs to be stored redundantly. Role-templates and -instances are stored in SCA Tool's database, while

```java
PermissionsCheckRequest req = new PermissionsCheckRequest();

// Metadata
var metadata = new PermissionCheckRequestMetadata();
metadata.setDepth(20);
req.setMetadata(metadata);

// Entity
var entity = new Entity();
entity.setType(objectType);
entity.setId(objectId);
req.setEntity(entity);

// Permission
req.setPermission(permission);

// Subject
var subject = new Subject();
subject.setType(subjectType);
subject.setId(subjectId);
req.setSubject(subject);

// Send request
PermissionCheckResponse response = permissionApi
    .permissionsCheck(tenantId, req)
    .blockOptional();

boolean canAccess = response.getCan() == CheckResult.ALLOWED;
```

**Listing 7.2:** Permify SDK permission-check API request example

the resulting classical roles, after translation through the authorization-service, are also stored in Permify's database for simple access-checks. Since the two databases are controlled from two different applications, with none of the applications having access to the other's database, there is no trivial way of keeping the data consistent in case of an error. As elaborated in Chapter 2.4, this is a common problem in microservice architectures. One possible solution to achieve inter-service consistency is to implement the saga pattern.

The choreography-based saga approach is adapted to keep SCA Tool's and Permify's database consistent. Usually the messages send via the Saga pattern could be modeled through Spring Boot events. The achieved benefit is the low coupling between the micro-services. In this case however the situation is simpler, since the participants of a saga are the authorization-serive and the internal permify-wrapper, only accessed through said authorization-service. Thus, function calls are sufficient.

While Spring Boot offers transactional functions, automatically rolling back made changes upon the occurrence of an error, this only works for SCA Tool's database.

Permify's database needs to be kept consistent manually. For better visalization, the previous representing setting or updating the permissions of a role (see Figure 7.2b) is modified to reflect this approach. Figure 7.6 show said modified version. As can be seen, changes made to Permify via the permify-wrapper need to be explicitly rolled back. More precisely, when an error occurs while setting the permissions of a role-instance via Permify, all previously updated role-instances are rolled back to their previous state. Thus, a basic form of consistency between both databases is ensured.

While this approach does offer basic synchronization between both databases, there are cases where the implementation cannot ensure consistency. A simple scenario is the execution of a transaction, including multiple calls to Permify, while SCA Tool permanently loses connection to the Permify instance during execution. If some calls to Permify have already been successfully made, while others are still pending, neither a successful transaction nor a rollback is possible. This leads to an inconsistent state between both databases, which the system cannot recover. Spring Boot events would not aid in this situation either, since they are not persisted until the Permify-instance or the whole system is restarted. A solution that would presumably solve this issue is the usage of persisting queues. Saga messages would be transfered via said queues, persisting until they are consumed. While the concept is rather simple, the integration is not trivial. Such an enhancement is relevant for future development, but was not pursued during this thesis.

### 7.2.8 Migration of Permify Schemas

Permify uses a schema, written in their modeling language, to define authorization rules or for SCA Tool the controlled types of resources and their available permissions. For SCA Tool, it contains the resources and permissions declared in Subsection 7.2.3. The schema file does not need to be statically provided to Permify, but is transferred to the running Permify instance via an API call. As other Permify-API calls, the written Permify-wrapper makes the call accessible to the backend. Hence, the backend simply passes a schema-file to the Permify-wrapper and Permify updates its internally stored representation.

While this is suitable for initializing the model, SCA Tool is developed continously, including changes in resources and their permissions. Since Permify does not offer a simple way to compare the currently active schema to the one present in SCA Tool, cleanly updating the schema is an issue. Consequently, a simple schema-initialization job is introduced, running once on each startup of SCA Tool.

It checks if a new schema-file was added to the backend, and if so tries to update Permify accordingly. If successful, SCA Tool persists the new version number as

current. If not, the error gets logged and the current schema is kept.

The approach eliviates unnecessary updating of the schema on Permify's side, which would add new entries to its database, even if the passed schema is already stored. It additionally adds transparency during development on what schema is currently used, which otherwise could not be easily determined. Since updates to the schema file are always made via a new file, changes made earlier can also be traced back, which simplifies debugging.

## 7.3 System Administration Dashboard

The administration of an application's users is a key necessity when providing a product. SCA Tool currently lacks such a feature. The only option system administrators currently have to view and manage users of SCA Tool, is to manually connect to and analyze the database. Consequently, a system administration dashboard is added to SCA Tool. To additionally allow for finer user management, the concept of user states is enhanced.

### 7.3.1 Enhancing User States

The existing implementation technically offers the following three user states: *Active*, *Waitlisted*, *Deleted*. While representing user states, they are not well implemented as such. The existing implementation is solely based on separate timestamps, representing when a user entered the waitlist or deleted their account. This implementation is error-prone and difficult to maintain. Thus, a better maintainable solution is implemented.

For larger systems with complex state switching, the state pattern[2] is well suited. It is an OOP-based approach, allowing subclasses to implement which functionality a state provides and how the functionality behaves in a state. For SCA Tool, the state pattern is not well suited. The defined states in SCA Tool are very lightweight, not needing this much control. The pattern additionally does not fit SCA Tool's current architecture. User-functionality is not implemented in OOP-based classes but in global services, working with anemic data objects.

A better suitable solution is to simply store a user's state in every user object. It adheres to the anemic domain model, with the user-service providing the necessary functionality to switch states. With this, a well maintainable and expressive representation of user states is implemented.

[2] https://refactoring.guru/design-patterns/state

### 7.3.2 Administration Dashboard

For the sake of simplicity and accessibility the dashboard was integrated into SCA Tools frontend. The dashboard shall only be accessible by administrators, with access from others being a high security risk. To mitigate possible misconfiguration, instead of using the previously introduced authorization workflow, the concept of system-administrators is introduced. Users deemed an adminstrator are intentionally not configureable at runtime, but are hardcoded into the backend. They have unique priviledge, which can be checked separately. This is done to secure administrator-specific endpoints against any access, but by explicitly defined users. They additionally have the priviledge of passing any authorization check, independent of stored permissions. This allows administrators to have full control of SCA Tool.

The system dashboard offers two main functionalities: User management and Organization management. A list of all current users can be viewed and navigated to see details on individual users. The detailed view allows to change the user's state, removing them from the waitlist or suspending a user. It additionally shows all organizations they are part of, and adds the option to remove them from an organization. Users can additionally be logged out of all active sessions and login to SCA Tool can be disabled.

Separately, all existing organizations may be viewed, with the option to navigate into an oranizations settings. This gives administrators full access to all organization management functionality which the organization owner posseses.

Concluding, the administration dashboard allows system administrators to manage SCA Tool's users and organizations.

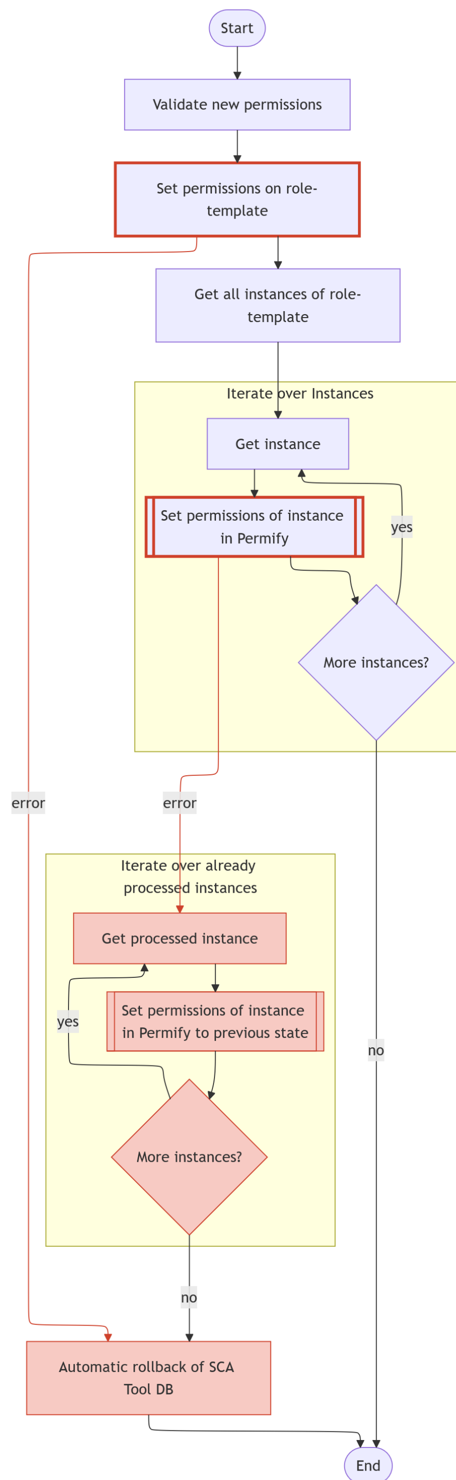



**Figure 7.6:** Handling consistency issues during role permission updating by manually rolling back Permify's data. Rollback processes are filled red.

# 8 Evaluation

The chapter evaluates if the requirements (defined in 4) were fulfilled during the work of this thesis.

## 8.1 Functional Requirements

This section evaluates the functional requirements.

### 8.1.1 Users

**F-1:** Fulfilled. Users can create private projects in their personal organization.

### 8.1.2 Addition of Organizations

**F-2:** Fulfilled. Organizations group multiple members, having shared access according to their authorization.

**F-3:** Fulfilled. Users can create organizations via the UI.

**F-4:** Fulfilled. Organization members with adequate authorization can invite other users to join the organization.

**F-5:** Fulfilled. A list of users is shown in the organization-settings. It additionally offers the removal of members from the organization.

**F-6:** Fulfilled. Organizations can be modified via the organization-settings.

### 8.1.3 System Administration

**F-7:** Not fulfilled. Integrating management of Kubernetes deployments into the dashboard is of higher complexity than initially expected. The management can be further done via Lens.

**F-8:** Fulfilled. A list of all SCA Tool users can be viewed in the dashboard.

**F-9:** Fulfilled. The details of a user account can be viewed in the dashboard.

**F-10:** Partially fulfilled. The login of users can be enabled and disabled. Disabling registration is not easily possible via Kratos.

**F-11:** Fulfilled. The dashboard offers the option to log individual users or all users out.

**F-12:** Fulfilled. The state of a user can be changed via the dashboard.

**F-13:** Fulfilled. Administrators can access the oranization-settings of all organizations through the dashboard.

**F-14:** Fulfilled. Removal of organizations is accessible via the organization-settings.

**F-15:** Not fulfilled. Monitoring of system resources for distributed deployments is of higher complexity than expected and was disregarded for the scope of the thesis.

**F-16:** Not fulfilled in this thesis. Sentry was introduced to SCA Tool during the thesis. It fulfills that requirement.

### 8.1.4 Access Control

**F-17:** Fulfilled. An access control system was integrated into SCA Tool.

**F-18:** Fulfilled. The frontend fetches the permissions of the user and only shows relevant UI elements.

**F-19:** Fulfilled. The organization-settings offer role-management functionality.

**F-20:** Fulfilled. Default roles are created for every organization.

## 8.2 Non-Functional Requirements

This section evaluates the non-functional requirements.

### 8.2.1 Performance

**NF-1:** Not fulfilled. Measurements on the local system deviated noticeably with the system load. Measurements on the deployment were not conducted.

**NF-2:** Partially fulfilled. Permify caches permission-checks, but the wrapper does not.

**NF-3:** Fulfilled. The frontend caches a users permissions for a session.

### 8.2.2 Portability

**NF-4:** Fulfilled. Permify is easily deployable via helm charts. An earlier version of the implementation was successfully deployed.

**NF-5:** Fulfilled. Permify offers a docker image for local development.

### 8.2.3 Maintainability

**NF-6:** Fulfilled. The access control system was adapted to the anemic layered architecture of SCA Tool's backend.

**NF-7:** Fulfilled. Permission-checks are permformed through a single function call.

**NF-8:** Fulfilled. Permify's schema is easily editable.

### 8.2.4 Reliability

**NF-9:** Fulfilled. All necessary endpoints provide the required authorization-checks.

**NF-10:** Fulfilled. If the Permify instance shuts down, no access is authorized.

### 8.2.5 Integrity

**NF-11:** Partially fulfilled. A variation of the saga pattern was implemented. It does handle most errors during inter-service transactions correctly. A consistent shutdown of Permify could still lead to inconsistencies.

# 9 Conclusions

With the work conducted in this thesis, organizations were added to SCA Tool. Consequently, multiple members now have the ability to work on shared resources. Since not all members of an organization shall have access to all resources, an access control system was integrated. It uses Permify as an authorization service, and adds a translation layer to offer SCA Tool a non resource-bound variant of role-based access control. Roles are implemented in a fully customizable way, allowing organizations of various hierarchies to represent the best suitable roles for their use case.

To allow system administrators to manage users and organizations, a system management dashboard was integrated into SCA Tool's frontend. It allows to view all users and their individual details, including organizations they are a member of. The state of users can be managed to move users from the waitlist, suspend them or delete their account fully. Users can additionally be logged out of the system, and the general login to SCA Tool can be disabled. The dashboard also offers organization management. All organizations in SCA Tool can be viewed and fully managed.

# References


Agent, O. P. (n.d.-a). *Comparison to other systems - OPA* [Open policy agent]. Retrieved March 19, 2025, from https://www.openpolicyagent.org/docs/latest/comparison-to-other-systems/

Agent, O. P. (n.d.-b). *Ecosystem - OPA* [Open policy agent]. Retrieved March 19, 2025, from https://www.openpolicyagent.org/ecosystem/

Agent, O. P. (n.d.-c). *Introduction - OPA* [Open policy agent]. Retrieved March 19, 2025, from https://www.openpolicyagent.org/docs/latest/

Agent, O. P. (n.d.-d). *Open policy agent*. Retrieved September 5, 2024, from https://www.openpolicyagent.org/

Agent, O. P. (n.d.-e). *Policy language - OPA* [Open policy agent]. Retrieved March 19, 2025, from https://www.openpolicyagent.org/docs/latest/policy-language/

Agent, O. P. (n.d.-f). *Rego playground - access control*. Retrieved March 19, 2025, from https://play.openpolicyagent.org/?example-group=access-control

Aydin, S., & Çebi, C. B. (2022). Comparison of choreography vs orchestration based saga patterns in microservices. *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, 1–6. https://doi.org/10.1109/ICECET55527.2022.9872665

Casbin. (n.d.-a). *How it works | casbin*. Retrieved March 17, 2025, from https://casbin.org/docs/how-it-works/

Casbin. (n.d.-b). *Overview | casbin*. Retrieved March 17, 2025, from https://casbin.org/docs/overview/

Casbin. (n.d.-c). *RBAC | casbin*. Retrieved March 17, 2025, from https://casbin.org/docs/rbac/

Casbin. (n.d.-d). *Supported models | casbin*. Retrieved March 17, 2025, from https://casbin.org/docs/supported-models

Casbin. (n.d.-e). *Syntax for models | casbin*. Retrieved March 17, 2025, from https://casbin.org/docs/syntax-for-models/

Cemus, K., Cerny, T., Matl, L., & Donahoo, M. J. (2016). Aspect, rich, and anemic domain models in enterprise information systems. In R. M. Freivalds, G. Engels & B. Catania (Eds.), *SOFSEM 2016: Theory and practice of*

*computer science* (pp. 445–456). Springer. https://doi.org/10.1007/978-3-662-49192-8_36

Ferraiolo, D., Cugini, J., & Kuhn, D. R. (1995). Role-based access control (RBAC): Features and motivations. *Proceedings of 11th annual computer security application conference*, 241–48.

GitHub. (n.d.-a). *Casbin basic example - GitHub* [GitHub]. Retrieved March 18, 2025, from https://github.com/casbin/casbin/blob/master/examples/basic_model.conf

GitHub. (n.d.-b). *Casbin examples - GitHub* [GitHub]. Retrieved March 18, 2025, from https://github.com/casbin/casbin/tree/master/examples

GitHub. (n.d.-c). *Casbin examples - GitHub* [GitHub]. Retrieved March 18, 2025, from https://github.com/casbin/casbin/blob/master/examples/rbac_model.conf

GitHub. (2025, March 13). *Ory keto - GitHub* [original-date: 2018-03-17T10:40:15Z]. Retrieved March 15, 2025, from https://github.com/ory/keto

Hu, V. C., Ferraiolo, D. F., & Kuhn, D. R. (2006). *Assessment of access control systems* (NIST IR 7316) (Edition: 0). National Institute of Standards and Technology. Gaithersburg, MD. https://doi.org/10.6028/NIST.IR.7316

OpenFGA. (n.d.-a). *Configuration language | OpenFGA*. Retrieved March 20, 2025, from https://openfga.dev/docs/configuration-language

OpenFGA. (n.d.-b). *Custom roles | OpenFGA*. Retrieved March 21, 2025, from https://openfga.dev/docs/modeling/custom-roles

OpenFGA. (n.d.-c). *Introduction to FGA | OpenFGA*. Retrieved March 20, 2025, from https://openfga.dev/docs/fga

Ory. (n.d.). *Ory keto - permission and role management*. Retrieved March 16, 2025, from https://www.ory.sh/keto/

Ory. (2025a, March 10). *Introduction to ory permissions | ory*. Retrieved March 15, 2025, from https://www.ory.sh/docs/keto

Ory. (2025b, March 10). *Ory permission language specification | ory*. Retrieved March 16, 2025, from https://www.ory.sh/docs/keto/reference/ory-permission-language

Ory. (2025c, March 10). *Versioning and upgrades | ory*. Retrieved March 15, 2025, from https://www.ory.sh/docs/ecosystem/upgrading

Permify. (n.d.-a). *Deploying permify with helm charts* [Permify docs]. Retrieved March 31, 2025, from https://docs.permify.co/setting-up/installation/helm

Permify. (n.d.-b). *Interacting with the API | permify* [Permify docs]. Retrieved March 21, 2025, from https://docs.permify.co/getting-started/enforcement

Permify. (n.d.-c). *Introduction | permify* [Permify docs]. Retrieved March 21, 2025, from https://docs.permify.co/permify-overview/intro

Permify. (n.d.-d). *Modeling authorization | permify* [Permify docs]. Retrieved March 21, 2025, from https://docs.permify.co/getting-started/modeling

Permify. (n.d.-e). *Run using docker* [Permify docs]. Retrieved March 31, 2025, from https://docs.permify.co/setting-up/installation/container

Samarati, P., & de Vimercati, S. C. (2001). Access control: Policies, models, and mechanisms. In R. Focardi & R. Gorrieri (Eds.), *Foundations of security analysis and design* (pp. 137–196). Springer. https://doi.org/10.1007/3-540-45608-2_3

Sandhu, R., & Samarati, P. (1994). Access control: Principle and practice [Conference Name: IEEE Communications Magazine]. *IEEE Communications Magazine*, *32*(9), 40–48. https://doi.org/10.1109/35.312842

SOPHIST GmbH. (2024). *Schablonen für alle Fälle* (6th ed.). Retrieved February 18, 2025, from https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/Wissen_for_free/MASTeR-Broschuere_Int/MASTeR_Broschuere_6-Auflage_31-07-2024_AvP_V4.pdf

Styra. (n.d.). *Styra* [Styra]. Retrieved March 19, 2025, from https://www.styra.com/