Permission Concept SAP Sailing Analytics

# Introduction

This document describes the permission concept developed for the SAP Sailing Analytics (in the following just Sailing Analytics). Currently a very rough permission system based on role based access control (RBAC) is used to e.g. restrict access to the administration console. The system is built on the Apache Shiro (in the following just Shiro) framework. This system currently does not support unified user management (in the sense of a central user management system that manages the users for all deployments of the Sailing Analytics) or dynamic access control for all aspects of the Sailing Analytics.

However, as one medium term goal is to develop the Sailing Analytics to be usable by sailing clubs and eventually individuals as a cloud application, the user management and access control should be unified and expanded to be dynamic and provide the appropriate (the Sailing Analytics manages very personal data) security aspects.

The concept developed in this document should focus on fine grained access control with a strong focus on being able to unify and strengthen the user management and access control. Resulting in a system where multiple organizations and individuals can work on one system without unwantedly interfering in their actions and data.

A secure access control concept for the existing data model is not easily developed, because the data objects in the Sailing Analytics do not merely form static trees. Data objects can form graphs where there is no clear root for a given node. Furthermore the associations of data objects can change. These challenges have to be addressed by possible concepts by adapting them to this very specific domain.

The following requirements result from the above described (the access control system has to…):

1. Be expressive enough to support the complex associations of the Sailing Analytics
2. Support multiple organizations (clubs, events and individuals) working in one system
3. Communicate the permissions to the frontend (so only UI elements that support permitted actions are active)
4. Be reasonably complex and implementation intensive

# Access Control Concepts

The two big concepts that play together in this permission concept are access control lists (ACLs) (also used e.g. in the Linux or Windows file system) and RBAC. Furthermore, there is the concept of attribute based access control (ABAC) that is not explored in this concept document.

The concept of ACLs is based on the idea of assigning each data object that is access controlled an ACL. The ACL is a list of entries that assign a user or group of users to permissions. If e.g. read access is requested for a data object, its ACL is checked if the user or a group, the user belongs to, has an entry granting the read permission.

RBAC is based on the idea of having roles that imply certain permissions and assigning roles to users. The roles of RBAC are on a simple level equivalent to groups for ACLs. (Barkley, 1997) However, in general roles combine a set of users with a set of permissions, whereas groups represent only a set of users. According to (Sandhu, Coyne, Feinstein, & Youman, 1996) there are multiple models for RBAC. The model described above is called . Furthermore, there are , and which all include , but add additional features.

adds the concept of role hierarchies, where roles inherit the permissions granted by their parent roles. However, they do not inherit the set of users. adds constraints which restrict how and when roles and permissions can be combined with other roles or permissions. Besides mutual exclusion of roles or permissions, constraints could also require a user to have role *s* when assigned role *r* (or the same with permissions, which could be used e.g. to require to be able to view an event when a view permission for a race in the event is granted). Furthermore, the concept of constraints could also restrict the roles and permissions that a user can simultaneously have in a session (e.g. only one tenant role at a time). combines and .

In the context of RBAC (Ferraiolo, Cugini, & Kuhn) mentions the concept of subjects. A user may in one session only have a certain set of roles and permissions. This may be due to choice to reduce accidental actions with a wrong role or a constraint enforced by the system (See ). This set of roles and permissions is called a subject. A user may have any number of active subjects. However, a subject is only associated with one user.

It is to note that simple RBAC models show no difference in their ability to express access control policies than ACLs. (Barkley, 1997) More complex RBAC models are more expressive than ACLs.

In the course of this concept document there will be no difference in meaning between roles and groups. In the existing system they are named roles, thus the term roles will be used for both roles and groups.

# Initial Idea

The initial idea was a system that fully relied on roles and tried to imply permissions over the associations of data objects. However, it was found that implying permissions poses problems when e.g. data objects are unlinked from each other (or associations are changed), in which case permissions are lost, because they can no longer be implied. Moreover, as stated above the associations between the data objects in the Sailing Analytics do not necessarily form a tree, which prevents implications to be made just along a hierarchy. We discarded this idea and developed another, described below.

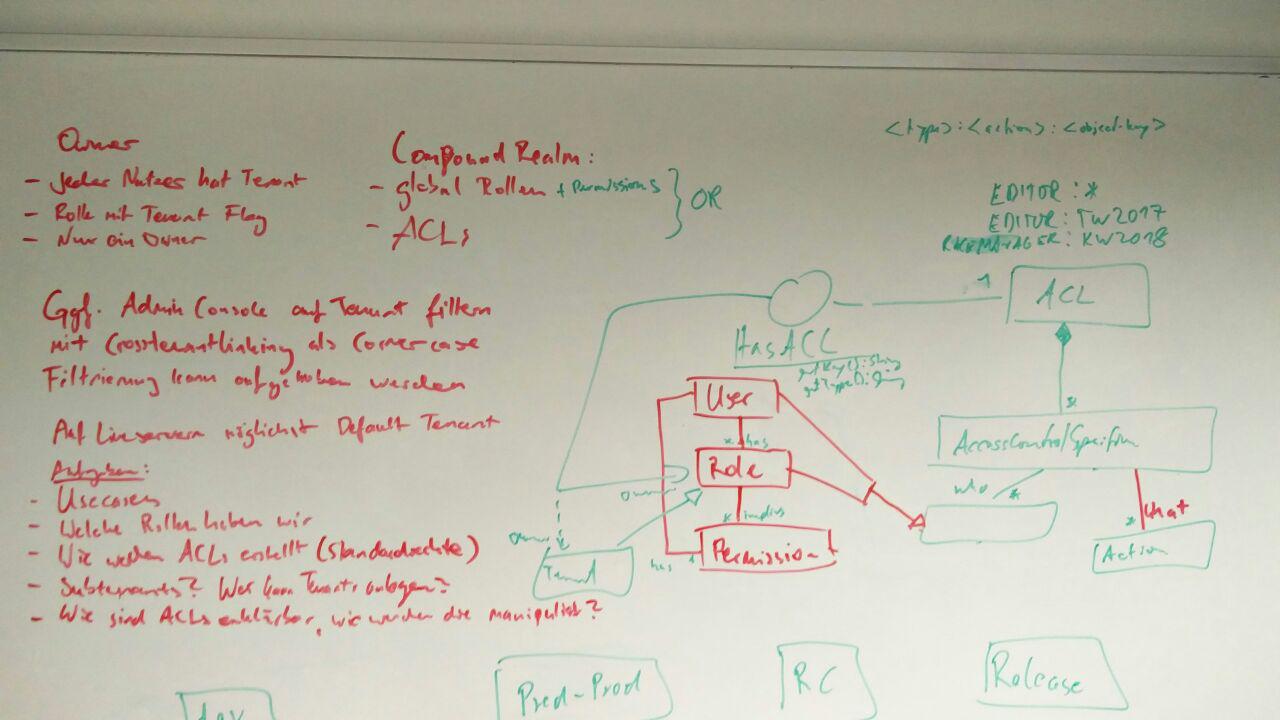


Figure 1 General concept (Walldorf 16.02.2017)

Figure 1 depicts the general initial ideas for this document that were developed after several iterations of concepts. The idea is based on the existing Shiro RBAC system that should handle global static roles and also supports directly granting permissions to a user. Furthermore, ACLs should be introduced. Those should solve the problem of “losing” implied permissions, because the access control lists are directly associated with the data object. The existing Shiro authorizing realm that checks for the roles and permissions directly assigned to a user would have to be extended by the ACL concept. This will form what we call a compound realm that if a permission is checked looks for the permission in the roles and permissions of the user and in the ACL of the data object. If either the roles and permissions or the ACL grants the permission the user is allowed access.

# Ownership

It is common place in cloud applications where multiple groups of users that each belong to some kind of organization work in one system to summarize these groups of users as tenants. The tenants represent the organizations and---if a hierarchy is allowed---the sub-organizations working in the system. In the Sailing Analytics the organizations could be SAP in general (e.g. archive server), sailing clubs, events like the Travemünder Woche or in the future private users.

One idea behind tenants is to encapsulate organizations so users of one organization cannot work with data objects from another organization if they are not granted the permissions explicitly. Furthermore, tenants are used to group data objects so users can have access to all data objects of a tenant and do not have to be granted every permission explicitly. Additional to data objects, tenants are also associated with roles. Thus, granting a role associated with a tenant is equivalent to granting the role for each data object which is associated with the tenant.

In the Sailing Analytics the set of roles for each tenant may, apart from certain exceptions, represent the subjects a user can adopt. A constraint could be introduced that only allows the roles for one tenant as a subject.

Tenants pose an UI problem, because it has to be clear to the user in which tenant he is currently working. The user has to be member of the tenant he is working for. Currently the best idea is to let the user select a tenant when he logs in and have default tenants for event and club servers that correspond to the event or club. Every following action is performed as that tenant.

In some cases, it might be necessary to transfer the ownership to another tenant/user. Thus, the owner should not be final but changeable.

## Users or Tenants as Owners

A problem with the tenant approach is that users could have no permissions to e.g. remove data objects that they have just created on accident, because the remove permission is reserved to admins of the tenant which a user that has create permissions may not be.

Three solutions come to mind. (1) The creator of a data object could always be granted the permission to remove the data object explicitly. (2) Moreover, the log where the creation was logged could be crawled to find the user that created the data object and override the permission system when in a certain timespan.

(3) Figure 2 shows an alternative approach to ownership. In this approach, a single user would own a data object so he can do everything with it. The tenant would then be a kind of secondary owner or group in Linux terms. This solves the problem that users could have no permissions to e.g. remove data objects that they have just created on accident. However, it also introduces a second layer of ownership. (4) Only grant the create permission when the user also has the corresponding delete permission.

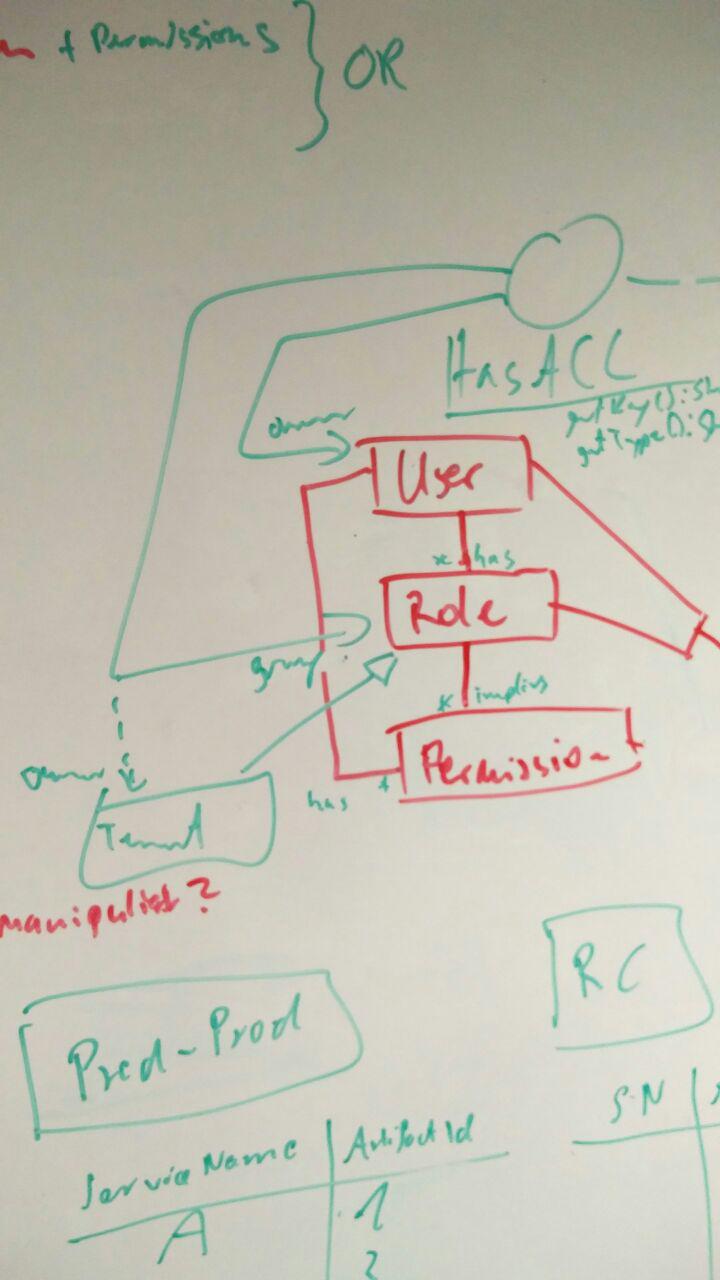
Approaches (2) and (4) are impractical. (2) is too complicated. The user would need delete permission for everything in the tenant for (4) to work.

Figure 2 Alternative concept for tenants and owners (Walldorf 16.02.2017)

Approach (1) solves the problem on hand, however these explicitly granted remove permissions are not just removed when the ownership of the data object changes, but remain. This could lead to users being able to delete data objects in other tenants, just because they created the object. Approach (3) more explicitly creates an ownership relation that can be edited and is thus chosen.

As the tenant is a data object itself, it also has an owner. The owning tenant of a tenant is the tenant itself.

## Subtenants

Subtenants could be a convenient way to restrict the permissions of certain users to only a part of a tenant’s domain. However, this introduces a hierarchy of tenants that brings with it its own challenges. Imagine there is a tenant “tw2017” and the 49er boat class races should not be manageable by the same race managers that can manage races of the other regattas. So “tw2017” would require a subtenant “other” and “49er” that encapsulate the 49er boat class and everything else from each other. Now if a permission is checked on an ACL, the ACL has to traverse the tenant hierarchy to find out if the user is part of a role for a parent tenant that grants the permission. However, the convenience of tenant hierarchies might be stronger than the traversal problem, because the hierarchy will probably never be deeper than one or two levels.

Another challenge with subtenants is how to communicate the concept to users. Which also makes it harder to imply with which tenant or subtenant a user is currently working.

An alternative strategy is just creating a completely new top level tenant for the 49er boat class races of “tw2017”. This would not introduce a hierarchy, but would require users that have roles for all boat classes to have their roles for both tenants instead of only the role for the parent tenant.

## Administration of Authorization

This section will discuss how it is determined if a user can grant or revoke a permission. Therefore, we define two rules:

* We define authority as power which has been legitimately obtained (Krishnan & Zimmer, 1991)
* We regard ownership as the starting point for delegation of authority (Krishnan & Zimmer, 1991)

With these two facts in mind, data objects must have a single user as the owner. However, as discussed earlier tenants are a second tier of ownership. Not every user that can create data objects in a tenant should automatically be the owner of every data object of the tenant, but in return should also not lose rights e.g. for removing an accidentally created data object on creation, because the tenant is the owner. Thus, in our approach the creator of a data object will be the owner and the tenant he is currently logged in to will be the owning tenant.

Another challenge after having a concept for ownership is the delegation of power. Not every user should be allowed to delegate his permissions to other users, thus there has to be a “grantPermission” permission that allows a user to delegate all his permissions to other users.

## Implementation of Ownership

Ownership is modeled as an explicit association between exactly one user and data objects and exactly one tenant and data objects. Owning a data object implies having all permissions on that object, as ownership is the regarded as the source of authority. In order to change ownership one has to be either owning user or tenant owner. If the user only changes the tenant owner he may remain owning user. If the user only changes the owning user he may remain owning tenant.

## Implementation of sharing data objects with public

In general, sharing a data object with the public should just be granting the “view” permission to everyone. This in return allows everybody that knows the link where one can view the data object to view it.

This is separated from promoting this public data object (e.g. event) on the official SAP site. This would have to be a separate list which can be edited by e.g. media admins. This would however only link to the source, because otherwise all promoted material would have to be imported to the archive.

# Permissions in the Frontend

Currently the permissions of the roles are hard coded and can thus be easily imported in the frontend. Dynamic roles that can change on runtime would require passing the permissions implied by the roles to the frontend.

One option would be to resolve all permissions of a user before passing the set of permissions into the frontend. In a distributed system with multiple servers where a user could have permissions this is no viable solution.

ACLs could be delivered with the object itself. A permission on an object can then be checked in the frontend by asking the ACL delivered with the object. This would require adding a call to the permission system to every remote procedure call that returns an object.

A third but possibly resource hungry possibility would be to implement a service that can be called from the frontend to check single permissions. The service would implement some kind of hasPermission(permission) method. This could then be used from the frontend as well as the server code.

As ACLs will probably remain small in general, we will implement the second approach. Furthermore, the ACLs that are returned by the server will be reduced to the entries that are relevant for the current user.

# Permission Defaults

This section will discuss how to handle default permissions. Default permissions for data objects should not be implied by the context they are created in. Implying permissions could lead to unwanted permissions on data objects.

There are two options for default permissions:

1. Each type of object has its own mask of permissions it gets assigned. This has the advantage over pure roles that if one wants to change the norm only for one instance of an object one can. The masks could even be editable for each tenant so different defaults can be set. However, this also requires that the default ACL contents are conveyed to the user for checking and display.
2. The creator can choose on creation which other permissions to grant.
3. Have default permissions in roles so they do not have to be entered into each ACL, but have negative permissions to revoke defaults.

The default permissions will be handled by approach 3. to keep the ACLs short and less redundant.

# Implementation of Roles

There currently are only a few hardcoded global roles. These shall be usable in the future too and should be independent of the server or the tenant the person that has this role is working on. These include:

1. Global Admin  
   Has permissions for everything
2. Create Tenant (Leute die Events/Server anlegen)
3. Media Admin

Furthermore, there would be a difference between global roles and roles used in ACLs. The most basic role used in ACLs is the tenant role that exists for every tenant. This role is only granted to a few people that have every right for every data object the tenant owns. The other roles in a tenant that are of the pattern “role:tenant” (where tenant is replaced by the tenants name) are custom to every tenant, but some examples are listed here:

1. Tenant Owner “owner:tenant”  
   Can delete the tenant, additionally to everything the tenant admin can do
2. Tenant Admin “admin:tenant”  
   Has (almost) every permission in his tenant
3. Eventmanager “eventmanager:tenant”
4. Racemanager “racemanager:tenant”
5. Editor “editor:tenant”
6. Resultservice “resultservice:tenant”

# Constraints

A problem that is not easily solved with either ACLs or RBAC is constraining accesses in a more complex way than checking for a permission. A use case that may be important in the future when clubs can use the Sailing Analytics on their own is as follows.

Clubs may only be able to create races with e.g. < 60 boats, so club events cannot exceed the infrastructure provided to them. How could this be implemented with permission checking?

A “create\_big\_race” permission could be hardcoded that is checked when a user tries to add more than 60 competitors to a race.

There are even more expressive access control systems than RBAC. They are called constraint based access control systems. They allow constraints to be expressed in a less black and white way, however are very complex. This concept is not supported by the permission concept proposed here, because use cases like the above are probably edge cases that will be hard coded.

# Use Cases

In the following example use cases are listed that describe how user actions will impact the ACLs of the data objects the user interacts with. It is to note that this list of use cases is no complete list of all use cases for the permission handling system. Listing all of them is outside the scope of this document.

It is always assumed that the ID of the user is “user” and the ID of its tenant is “tenant”.

1. Create Event (or any other data object)
   1. User creates event
   2. Event is owned by tenant that user is associated with in this session
   3. Access control list is created for the event
   4. Permissions as e.g. “view” are implicitly granted by the roles of the owning tenant
2. Transfer ownership of event (or any other ownership transfer) (Already described in section “Ownership”)
3. Link RegattaLeaderboard into LeaderboardGroup
   1. If either the user, a role or a tenant the user is part of, has the permission to edit the LeaderboardGroup (LBG) and view the RegattaLeaderboard (RL), the user can link them.  
      LBG ACL = {“user”:[“edit”]}  
      RL ACL = {“user”:[“view”]}
   2. If the user has the “grantPermissions” permission, the “view” permission will automatically granted to all that can view the LeaderboardGroup.
4. Unlink TrackedRace
   1. If either the user, a role or a tenant, the user is part of, has the permission to edit the Leaderboard, the user can unlink them.
5. Share TrackedRace
   1. If either the user, a role or a tenant, the user is part of, has the permission to view the TrackedRace and the “grantPermissions” permission, the user can grant view permissions to anybody  
      ACL = {“user”:[“view, grantPermissions”]}
   2. The user shares the TrackedRace with “user2”  
      ACL = {“user”:[“view, grantPermission”], “user2”:[“view”]}
6. Create GPSFix
   1. It would only be consistent to attach a ACL to each GPSFix, however it probably never happens that a GPSFix has other permissions than a whole track, thus I propose to leave GPSFixes without ACLs and only introduce access windows on the tracks that can have their own ACL.
7. Masterdata import
   1. This should import all permissions as they are. A masterdata import itself is no reason to change permissions, however only data objects that the user that is importing has a “view” permission for should be importable. If the importing user is the owner or a tenant owner of the imported data, he can also change the ownership.
8. Share event with public
   1. If either the user, a role or a tenant, the user is part of, has the permission to view the data object and the “grantPermissions” permission, the user can share the event with the public  
      ACL = {“user”:[“view, grantPermissions”]}
   2. The user shares the data object  
      ACL = {“user”:[“view”], “\*”:[“view”]}
9. Revoke permissions
   1. Simply revoke: If either the user is the owner of or the user is a tenant owner/admin of the data object, the user can revoke every permission to the data object from anybody.
   2. Overwrite with negative: the same rules apply as for revoking, however a permissions in the following form is inserted into the ACL

{“user”:[“!view”]}

1. First boot of server
   1. On first boot of the server, an admin user is created. The creator of the server will log in as that admin user and in most cases create a new tenant. Thereafter, the creator will create at least one new users, assign the admin role to that user and delete the default admin user.

# Algorithm bool hasPermission(String permission) for Composite Realm

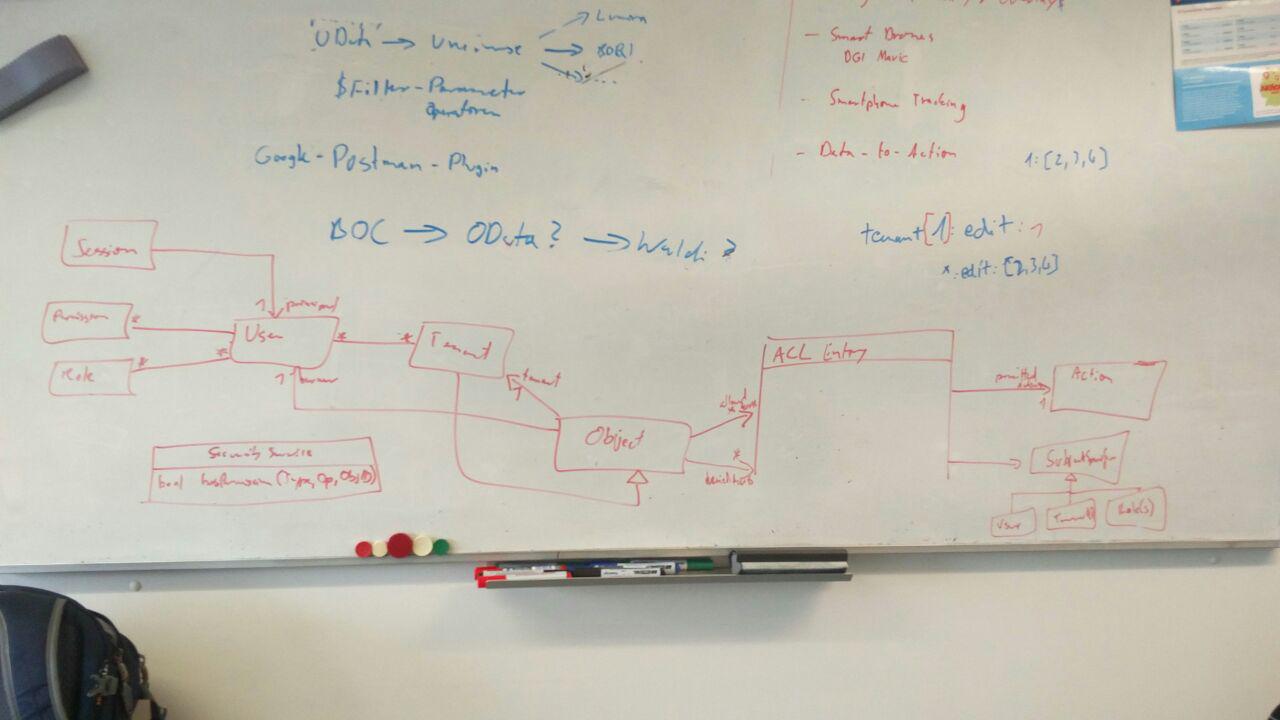


Figure 4 UML diagram

The above describes the data model that is relevant to the composite realm that implements the hasPermission function. The “permission” parameter should be of the pattern “type:action:instance”. It is assumed that the user (with associated permissions and roles), tenant, ownership associations and ACL entries are available. The following describes in which order the different sources for permissions are checked and how they depend on each other.

1. Check if the user is the owner or tenant owner of the data object for which the permission is requested
   1. If this is true return true
2. Check if the ACL entries grant or explicitly revoke the permission to the user under consideration of the user’s roles
   1. If there is an entry, return true if granted and false if revoked, but take the most explicit entry and in doubt return false
3. Check if the permission is directly assigned to the user
   1. If this is true return true
4. Check if a role grants the permission to the user
   1. If this is true return true

# Migration

With such an extensive existing system as the Sailing Analytics Suite, migration is a big concern. The existing RBAC system is easily extended to support ACLs. However, implementing permission checking in the whole system will be a long process, because probably almost every service request will have to be edited.

Another challenge besides the code changes is the data migration. For every existing data object an ACL has to be created and filled with the right permissions so the users do not notice a big change.

Besides creating an ACL for every data objects that is access controlled, an owner has to be defined for each existing data object, so that in combination with the ACLs no user loses permissions they need to have. In order to do this, where will have to be a script that associates all data objects on a server with a tenant and a specific user as owner (e.g. on the archive all data objects are associated with the tenant “archive” and owned by the user “Axel”). Thereafter, the users on the server are assigned their respective role.

# Bibliography

Barkley, J. (1997). Comparing Simple Role Based Access Control Models and Access Control Lists. (p. 6). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.6366.

Ferraiolo, D. F., Cugini, J. A., & Kuhn, R. D. (n.d.). Role-Based Access Control (RBAC): Features and Motivations.

Krishnan, I., & Zimmer, W. (1991). Delegation of Authority. *IFIP TC6/WG6.6 Second International Symposium*, (pp. 595-606). Washington DC.

Sandhu, R. S., & Pierangela, S. (1994). Access Control: Principles and Practice. *IEEE Communications Magazine*, 40-48.

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). Role-Based Access Control Models. *IEEE Computer, Volume 29, Number 2*, 38-47.