Permission Concept SAP Sailing Analytics

# Introduction

This document describes the permission concept developed for the SAP Sailing Analytics. Currently a very rough permission system based on role based access control (RBAC) is used to restrict access to for example the administration console. The system is built on the Apache Shiro framework. This system currently does not support unified user management and access control for all Sailing Analytics instances. However, as the Sailing Analytics should be pushed to be usable by sailing clubs and eventually individuals as a cloud application, the user management and access control should be unified.

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

A 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.

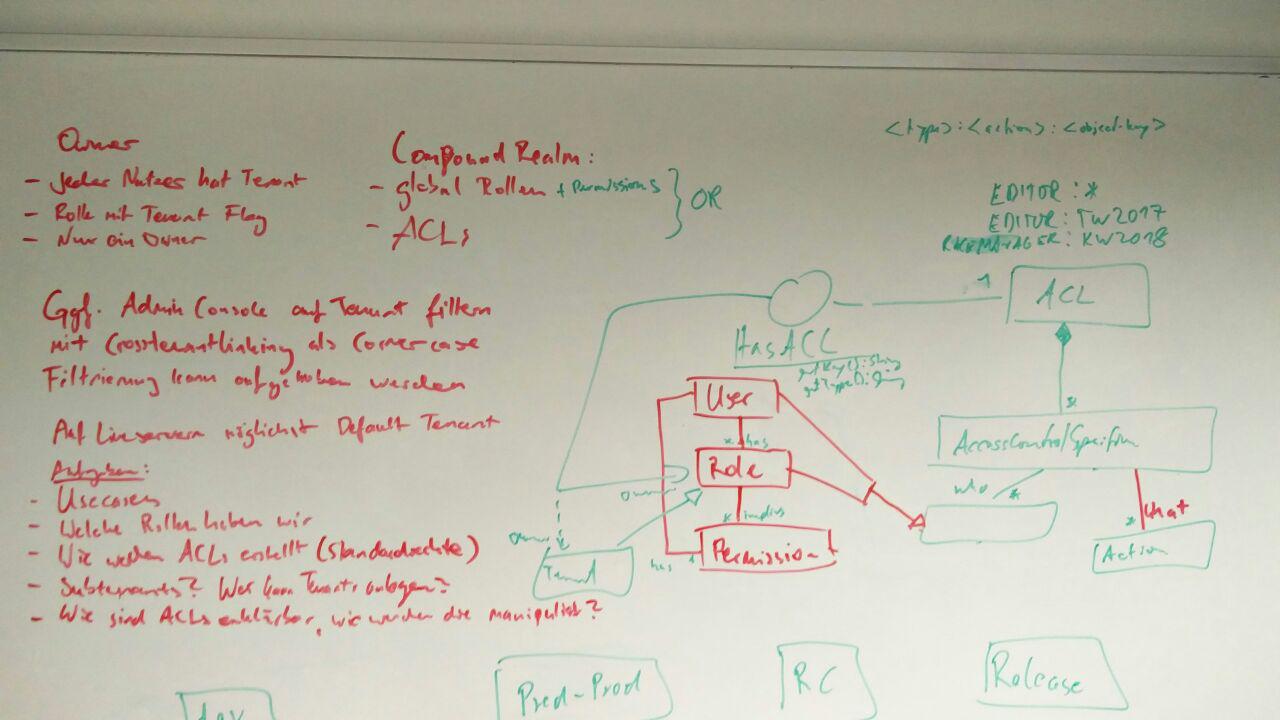


Figure 1 General concept (Walldorf 16.02.2017)

Figure 1 depicts the general ideas that were developed after several iterations of concepts. One 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 for example 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. The graph like structure also hamper understanding the implied permissions. The users of the system should not be required to read a manual to use it.

As depicted in figure 1, the idea further developed in this concept document is based on the existing RBAC system that should handle global roles and also supports directly granting permissions to a user. Furthermore, access control lists (ACLs) (also used e.g. in the Linux or Windows file system) should be introduced. Those should solve the problem of losing permissions that are implied, because the access control list stays on the data object. As the supporting framework Apache Shiro is used and should be used in the future too. The existing 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 granted the permission.

# Access Control Concepts

The two big concepts that play together in this permission concept are ACLs 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 user to permissions. If for example read access is requested for a data object, its ACL is checked if the user or a group the user wanting to have read access belongs to has the read permission.

RBAC is based on the idea of having roles that imply certain permissions and assigning roles to users. Furthermore, for special cases there is the option of assigning permissions directly to users. The roles of RBAC are on a simple level equivalent to groups for ACLs.

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 can be more expressive than ACLs.

# 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 dynamic creation of roles.

However, ACLs have to be deeply integrated into the system, because at least an interface that flags every data object that is access controlled has to be added to these objects. Furthermore at least an identifier for the ACL has to be exported to the frontend with every data object that is access controlled. It would be even better to export the ACL for every access controlled data object. This includes the groups of the user, because part of the ACL could link roles to users which would have to be resolved. If the ACLs are passed to the frontend with the data objects they belong to, the ACLs would need to be passed to their data objects on server startup when they are loaded from the database or retrieved in every service call that need to pass an ACL to the frontend, which would touch even more code.

If the solution to the above described frontend to backend interface challenge is to have a service that can be asked for the ACLs, it would be easier to implement a service that entirely checks permissions in the backend and only passes a Boolean result to the frontend. This would unify permission checking in the front- and backend and also simplify the handling of roles and ACLs.

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.

# Alternative Approach

An alternative approach would be to do everything the ACLs should be used for with roles, but instead of using heavily parameterized roles/permissions (that include a path to the data object as already mentioned in the introduction) use permissions that are only parameterized with an identifier to the instance they point to. So a role “eventmanagertw2017” could imply the permissions in the form “race:edit:tw2017:abc” (data\_object:action:tenant:race\_identifier) to all the races in the tenant and if a special case has to be implemented, a race could be removed from the role.

This would require no knowledge of the associations of the races and would not imply any other permissions. The system would be of equivalent power as ACLs, would require no double permission checking for roles and permissions on users and ACLs and would at least make it easier to migrate the code to the access control system.

# Data Model and Permissions (WIP)

This section lists the most important data objects and their associated permissions.

* Event
  + Read
  + Edit
  + Remove
* Regatta
  + Read
  + Edit
  + Remove
* Regatta leaderboard
  + Read
  + Edit
  + EditScores
  + EditCompetitors
  + ConfigureURL
  + Remove
* Race
  + Read
  + Edit
  + Remove
* Competitor
  + Read
  + Edit
  + Remove
* Device configuration
  + Read
  + Edit
  + Remove
* User
* Role

## 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 to pass the permissions implied by the roles to the frontend or the roles to be resolved before passing the set of permissions assigned to a user into the frontend.

ACLs are just 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.

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.

## Granting and Revoking Permissions (WIP)

This section will discuss how it is determined if a user can grant or revoke a permission.

# Tenant

It is common place in cloud applications where multiple groups of users that each belong to some kind of organization to summarize these groups of users as tenants. The tenants represent the organizations working on the system. In the Sailing Analytics the organizations could be sailing clubs or events like the Travemünderwoche. 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. Another idea is that data objects created in the domain of the tenant are not owned by a user that can leave the tenant or be unavailable, but are owned by the tenant itself.

Tenants pose an UI problem, because it has to be clear to the user in which tenant he is currently working. One solution would be 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.

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

## Users or Tenants as Owners

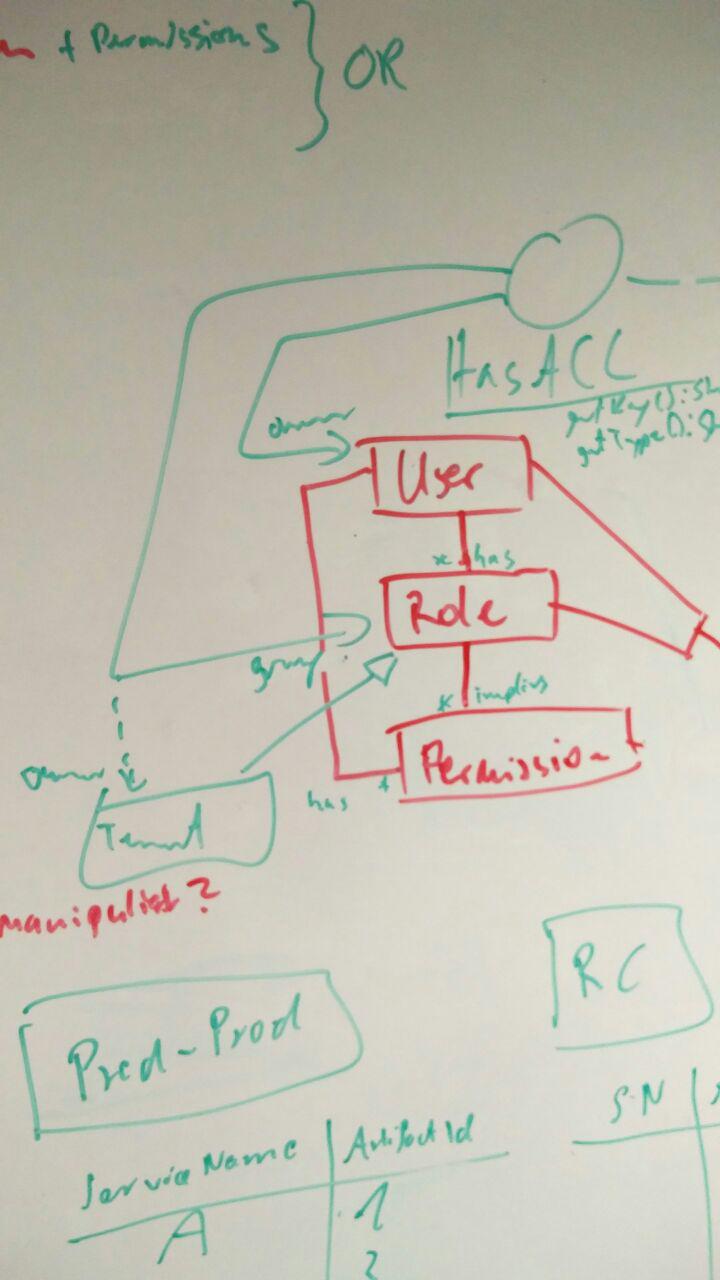


Figure 2 Alternative concept for tenants and owners (Walldorf 16.02.2017)

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 for example remove data objects that they have created on accident. However it also introduces a second layer of ownership. The remove permission for the creating user could also be entered into the ACL when it is created.

## 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 problems. Imagine there is a tenant “tw2017” and the 49er boat class races should not be manageable by the same Racemanagers 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.

Just creating a completely new tenant for the 49er boat class races 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.

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.

# How ACLs are created

ACLs could either be created when the associated data object is created or when they are used the first time, i.e. a first entry is created in the ACL. However, if the ACL is initialized with default permissions, the ACL would be created when the data object is created anyway. As data objects should only have one ACL each, the user should not be able to create the ACLs, this should instead be automated.

Especially for mass objects (GPSFixes) ACLs are only relevant when they contain entries, otherwise they are a waste of Java instances and memory space.

There are two options for default permissions:

1. Each type of ACL 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 he can. The masks could even be editable for each tenant so different defaults can be set.
2. Default permissions are set in the roles and only special cases are implemented via the ACLs. This means less entries in the ACLs and less work when creating and initializing the ACLs.

Default permissions for data objects should probably not be implied by their context they are created in. Implying permissions could lead to unwanted permissions on data objects.

# Roles (WIP)

Roles are currently hardcoded, however they need to be editable on runtime if they should also be used as groups in the ACLs. This implies that the UserDTO cannot obtain permissions for certain roles, only the hard coded ones. Furthermore there would be a difference between global roles and roles used in ACLs. The following roles are not necessarily global. Only the ones flagged as global are. The rest are abstract roles that every tenant should have.

1. Global Admin  
   Has permissions for everything
2. Global Moderator
3. Tenant  
   Is owner of everything in the tenants domain
4. Tenant Admin  
   Has (almost) every permission in his tenant (Maybe only users with the tenant role can delete the tenant, if this is possible) (One would not want to write the tenant admin role into every ACL when he has all the permissions anyway. This is a good example when we would want to imply permissions from a role that are not global)
5. Eventmanager
6. Racemanager
7. Editor?
8. Resultservice / Competitorcleanup

## The Role of Roles

Roles could either be used as a way to define global permissions for a small amount of users, like moderators and global admins. On the other hand they could also define default permissions when they are parameterized with tenant. (e.g. role “editor:tw2017” implies permissions “race:view,edit,delete:tw2017:\*”, “regatta:view,edit,delete:tw2017:\*”, …) Also see “How ACLs are created” 2. Option (Defaults).

## Tenants as Roles (WIP)

On the other hand not everybody who has a role for a tenant (e.g. “editor:tw2017”) has to also have the role “tw2017”, which would make him part of the tenant. The tenant would be the owner, but people who have permissions in the domain of the tenant would not have to be part of the tenant. Thus, a small group of people would make up the tenant, thus being the owner and a larger group would have permissions on the data objects of the tenant.

## Role Hierarchies (WIP)

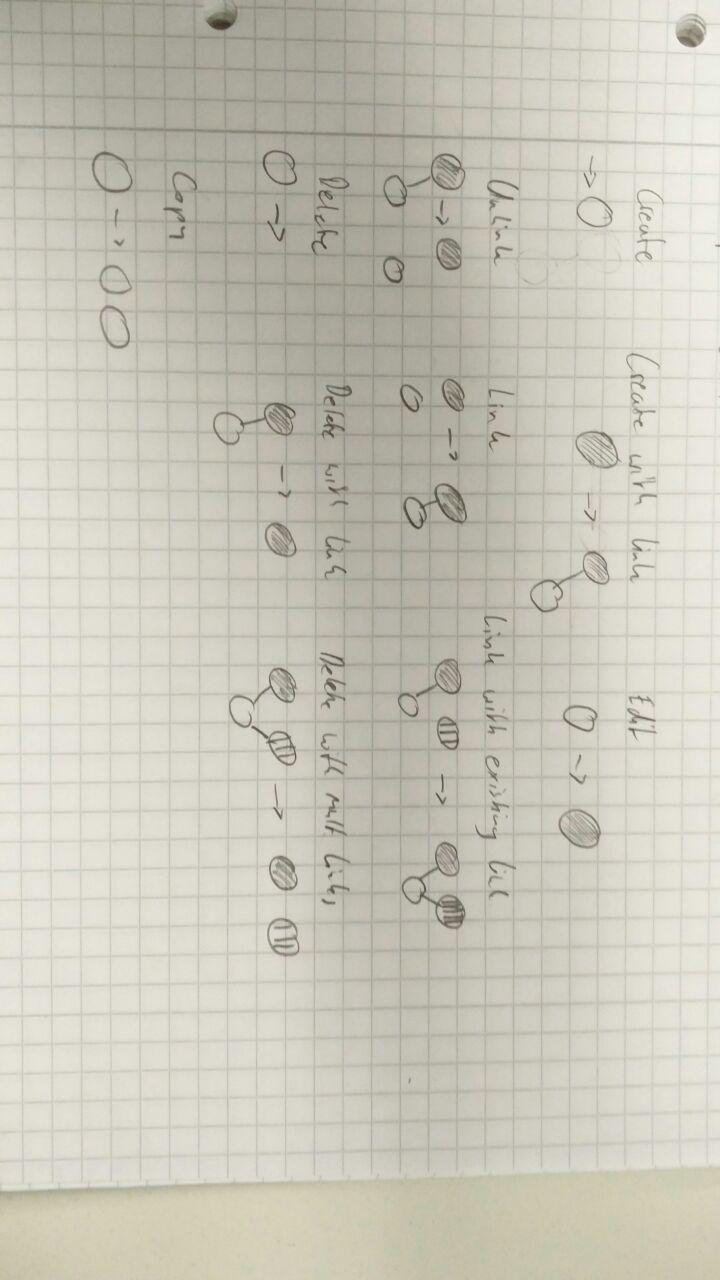
Here should be a discussion about whether to allow role hierarchies or not to allow them.

# Use Cases (WIP)

1. Create Event (or probably any data object)
   1. User is associated with a tenant
   2. User creates event
   3. Event is owned by tenant that user is associated with
   4. Access control list is created for the event
   5. View permission is set for all the users in the tenant   
      ACL = {“tenant”:[“read”]}
   6. Edit permission is set for all the tenant admins and event managers  
      ACL = {“tenant”:[“read”],   
      [“admin:tenant”, “eventmanager:tenant”]:[“edit”]}
   7. Remove permission is set for all the tenant admins   
      ACL = {“tenant”:[“read”],   
      [“admin:tenant”, “eventmanager:tenant”]:[“edit”],   
      ”admin:tenant”:[“remove”]}
2. Transfer ownership of Event
3. Link RegattaLeaderboard into LeaderboardGroup
4. Unlink TrackedRace
5. Share TrackedRace
6. Create GPSFix
7. Masterdata import
8. Share event with archive/public
9. Restrict users in a tenant from accessing an event
10. Granting a permission to a role
11. Revoking a permission from a role

# Possible Operations (very WIP)

Maybe ACL manipulations can be generalized to the operation, independent of the data object type. (There are more than found in the figure below. Might have to expand the figure)



1. Create
   1. Create an ACL
   2. Apply some kind of default permissions
2. Create with link
3. Edit
4. Unlink
5. Link
6. Link with existing link
7. Delete
   1. Delete ACL
8. Delete with link
   1. Delete ACL
9. Delete with multiple links
   1. Delete ACL
10. Copy

# Bibliography

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