

Managing collaborators on QDAcity

MASTER THESIS

Bharathwaj Ravi

Submitted on 3 June 2022



Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Dr. Andreas Kaufmann
Prof. Dr. Dirk Riehle, M.B.A.



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 3 June 2022

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, 3 June 2022

Abstract

Collaborative software allows people to work together to achieve a common goal. However, users might be distracted from their goals when such an application evolves with more functionalities. Moreover, when multiple users collaborate in the same session, the action of one user might confuse and come as a surprise to other users. These potential unexpected interactions between the users and the system contribute to the dynamic nature of the collaborative software.

For effective collaboration, a system should be able to minimize such unexpected interactions and enforce the expected behavior among the users. Access control is a standard data security practice that implements certain rules over the resources and users, thereby controlling the interaction.

In this thesis, We demonstrate an architectural refactoring of the access control for better collaboration in the example of QDAcity. QDAcity is a web application for conducting Qualitative Data Analysis (QDA) of text data. Researchers and students use this software to analyze qualitative data (such as interview data) gathered for scientific studies. In addition, it enables them to work on their research and share their findings collaboratively.

For QDAcity, being a collaborative application, resource management and data security is very important. Currently, QDAcity provides the same access rights to all the users of a project. In this thesis, we extend QDAcity's access control capabilities within a project by implementing a Role Based Access Control (RBAC) mechanism with fine-grained permission levels on the resources.

Contents

1	Introduction	1
1.1	QDAcity	1
1.2	Problem statement	2
1.3	Objective	2
2	Information Security and Access Control	3
2.1	Terminology	3
2.2	Types of access control	4
2.2.1	Discretionary Access Control (DAC)	4
2.2.2	Mandatory Access Control (MAC)	4
2.2.3	Attribute Based Access Control (ABAC)	4
2.2.4	Role Based Access Control (RBAC)	5
3	Requirements	11
3.1	Functional Requirements (FR)	11
3.1.1	Member management	11
3.1.2	Access control	13
3.2	Non Functional Requirements (NFR)	14
4	Architecture	17
4.1	Member management	17
4.1.1	Backend	17
4.1.2	Frontend	20
4.2	Access control	22
4.2.1	Backend	23
4.2.2	Frontend	23
5	Implementation	25
5.1	Member management	25
5.1.1	Backend	25
5.1.2	Frontend	37
5.2	Access control	38

5.2.1	Backend	38
5.2.2	Frontend	39
6	Evaluation	41
6.1	Functional Requirements	41
6.1.1	Member management	41
6.1.2	Access control	45
6.2	Non Functional Requirements	45
7	Future Work	49
7.1	Extending RBAC to other entities	49
7.2	Custom roles	49
7.3	RTCS integration	49
8	Conclusion	51
	References	53

List of Figures

2.1	Flat RBAC model (Sandhu et al., 2000)	6
2.2	Hierarchical RBAC model (Sandhu et al., 2000)	7
2.3	Hierarchical role tree	7
2.4	Constrained RBAC model with SSoD (Sandhu et al., 2000)	8
2.5	Constrained RBAC model with DSoD (Sandhu et al., 2000)	8
2.6	Symmetrical RBAC model with SSoD (Sandhu et al., 2000)	9
2.7	Symmetrical RBAC model with DSoD (Sandhu et al., 2000)	9
3.1	FunctionalMASTeR template	11
3.2	ISO/IEC 25010 - Categories	14
3.3	PropertyMASTeR template	14
4.1	Phases in member management of project	17
4.2	Unified Modelling Language (UML) class diagram of <i>Role</i> inheritance	18
4.3	UML class diagram of <i>Member</i> inheritance	19
4.4	<i>Member</i> factory UML diagram	19
4.5	Presentation and container components	21
4.6	Provider - Consumer pattern by React Context	24
5.1	UML class diagram of <i>Role</i>	26
5.2	UML class diagram of the permission enums	27
5.3	UML class diagram of <i>Member</i>	28
5.4	UML class diagram of the role enums	29
5.5	Project roles hierarchy	29
5.6	<i>Member</i> builder UML diagram	31
5.7	Additive role model	34
5.8	Migration workflow of data to RBAC model	37
5.9	Components hierarchy of container and presentation components .	38
5.10	Authorization flowchart for RBAC	38
5.11	Polymorphic authorization methods	39
6.1	Add User dialog	42

6.2	Add user group dialog	43
6.3	RBAC project members list User Interface (UI)	43

List of Tables

3.1	Project permissions and it's description	12
3.2	Access control matrix of project roles	12
5.1	Properties of <i>BaseRole</i> class	27
5.2	Properties of <i>BaseMember</i> class	30
5.3	RBAC properties of <i>Project</i> class	32
5.4	Permission and bitmask value	35
5.5	Access control matrix roles with effective permission	35
6.1	Lines of Code (LoC) coverage for unit test	47

List of Codes

4.1	Explicit access control for Owner role	22
4.2	Implicit access control for permission A	22
4.3	Explicit access control for Owner and Editor roles	22
5.1	Access control in frontend without containment component	39
5.2	Access control in frontend with containment component	40

Acronyms

FR	Functional Requirements
NFR	Non Functional Requirements
QDA	Qualitative Data Analysis
MAC	Mandatory Access Control
DAC	Discretionary Access Control
ABAC	Attribute Based Access Control
RBAC	Role Based Access Control
PoLP	Principle Of Least Privilege
SoD	Separation of Duties
SSoD	Static Separation of Duties
DSoD	Dynamic Separation of Duties
RA	Role Assignment
PA	Permission Assignment
RH	Role Hierarchies
SoC	Separation of Concerns
IP	Internet Protocol
API	Application Programming Interface
MVC	Model View Controller
DOM	Document Object Model
UI	User Interface
APD	Access Permission Data

JDO	Java Data Objects
UML	Unified Modelling Language
CUD	Create, Update, Delete
CRUD	Create, Retrieve, Update, Delete
MA	Member Assignment
DB	Database
CI	Continuous Integration
RTCS	Real-Time Collaboration System
GAE	Google App Engine
LoC	Lines of Code
E2E	End-to-End

1 Introduction

Collaborative software has become an integral part of our daily personal and professional life. However, Edwards (1996) stated that "*collaborative systems are potential chaotic environments*". Compared to single-user systems, multi-user collaborative systems have potential uncertainty and unpredictability because of the dynamic interaction between the users and the system. Furthermore, as the application evolves, users might get overwhelmed with the functionalities and get distracted from their intended goals. Therefore, a system should be able to control collaboration by ensuring the data is accessible only to users who need it, thereby minimizing chaos, unpredictability, and distraction of the users. One common way to control collaboration is by adopting the Principle Of Least Privilege (PoLP), according to which users should only be given minimum required access to accomplish their work.

In this thesis, chapter 1 presents the current state of QDAcity and the objective. Chapter 2 discusses information security and access control, followed by chapter 3, which discusses functional and non-functional requirements. In chapter 4, the architectural design of the backend and frontend is proposed, while chapter 5 describes how to put the design and architecture into practice. Chapter 6 evaluates the requirements. Finally, in chapter 7, we lay out potential future works before deriving the conclusion of this thesis in chapter 8.

1.1 QDAcity

Mezmir (2020) stated that QDA is "*concerned with transforming raw data by searching, evaluating, recognizing, coding, mapping, exploring and describing patterns, trends, themes and categories in the raw data, in order to interpret them and provide their underlying meanings*". The outcome of such an analysis is usually a theory about the studied phenomenon. QDAcity is a web application for conducting QDA. Projects can be created in QDAcity, where the data is gathered along with codes which are categorical labels to represent the data. Codes can be arranged in a hierarchical structure to form a code system. The process of assigning codes to data is called coding. Several researchers can collaborate on

projects to analyze and evaluate the data by coding it. Projects can have revisions which are a snapshot of all data, the code system, and all codings applied to the segments of data. A clone of the revision can be created with all the data and code system except for the coding applied. Researchers can recode in their cloned version of the project's revision. The project owner can create agreement reports to evaluate the inter-coder agreement between the recoded project and the associated project revision.

1.2 Problem statement

Currently, all project users in QDAcity can perform any actions within the project. It empowers them with too many privileges even if they are not intended to have them.

1.3 Objective

The following defines the main objective of this thesis.

Design and implement PoLP for effective controlled collaboration in QDAcity

2 Information Security and Access Control

This chapter discusses the concepts of information security, access control and their associated models.

According to the Computer Security Resource Center of NIST¹, information security is "*the protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability*".

2.1 Terminology

Common terminology related to information security is described below.

Access

Access is the ability of a subject to do an action on a particular object.

Authentication

Authentication is a technique by which the user's identity is verified. It can be performed in many ways and the most widely used methods are password-based and token-based authentication. Using such methods, when a user's identity is verified correctly, the user is said to be an authenticated user. In simple words, authentication answers, "who are you?".

Authorization

Authorization is a process that determines whether authenticated users can access or do an action on a given object. When access is given, the user is said to be an authorized user. In simple words, authorization answers, "what resources are you allowed to access?".

¹ <https://csrc.nist.gov/glossary/term/INFOSEC>

Confidentiality

It refers to hiding data from unauthorized access and allowing only access to the authorized users.

Integrity

It refers to the prevention of data against unauthorized modification.

Availability

It refers to the accessibility of data whenever it is needed to the users by protecting the system against denial-of-service attacks.

Access control

Ferraiolo et al. (2003) stated that "*Access control is concerned with determining the allowed activities of legitimate users, mediating every attempt by a user to access a resource in the system*".

2.2 Types of access control

Some of the most common types of access control models are described below.

2.2.1 Discretionary Access Control (DAC)

In the DAC model, the data owner decides to whom the data access should be given. It is very flexible as the owner can share ownership with other users. However, it is less secure because the initial owner cannot control the data sharing of other users. So it is not a good model for QDAcity.

2.2.2 Mandatory Access Control (MAC)

In the MAC model, the system determines to whom the data access should be given based on the security level of the data. For example, the security level of the data can be "Secret" or "Confidential". Users who have these security levels are allowed to access the data. It is an extremely secure model where administrators have high control over the data flow. However, it is less flexible because of the strict security labels check. So it is not suitable for QDAcity.

2.2.3 Attribute Based Access Control (ABAC)

In the ABAC model, the system decides to whom the data should be accessible based on the attributes like location, Internet Protocol (IP) address. Any required attributes can be configured and checked at run time, making it a very secure and flexible model. However, it is very complex to implement. Moreover, QDAcity

currently does not require authorization based on any such attributes, so it is not a good fit for now.

2.2.4 Role Based Access Control (RBAC)

In the RBAC model, the system decides to whom the data should be accessible based on the subject's assigned role. It is highly scalable as most of the authorization is done based on the predefined roles. It is also highly flexible, as the role's permission can be easily assigned or revoked. Compared to MAC and DAC, it supports data integrity by preventing unauthorized data modification. So based on these criteria, RBAC is the best-suited model for QDAcity.

According to Sandhu et al. (1996), three well-known security principles supported by RBAC are,

- **PoLP**: RBAC assigns only needed permission for the role, thereby restricting unwanted permission assignment
- **Separation of Duties (SoD)**: RBAC ensures no users should be assigned conflicting permission. For example, a user who applies for leave should not have permission to approve it
- **Data abstraction**: RBAC hides the data from the users based on their role's permission.

The most important terminologies related to RBAC are,

- **User group(UG)** refers to the collection of users who works to achieve a common goal
- **Subject(S)** in RBAC refers to an entity that requires access to an object. In the case of QDAcity, subjects are users or user groups
- **Object(O)** refers to the particular resource the subject wants to perform a specific operation. An example of this would be a specific document of a project in QDAcity
- **Action(A)** refers to a specific operation done on an object by a subject
- **Permission(P)** is the granular level of policies that defines what actions can be performed on which objects
- **Roles(R)** are common job titles that refer to the access level of the subject
- **Role Assignment (RA)** refers to assigning roles to subjects(users or user groups).
- **Permission Assignment (PA)** refers to assigning a set of permissions to a role. A role can have multiple permissions, thus allowing the subject to

perform different actions on objects through a single role

- **Role Hierarchies (RH)** refer to the inheritance within the roles. A role higher in the hierarchical tree inherits all the roles below it, thus inheriting all their permissions in addition to their own permissions
- **Member** in QDAcity refers to a subject with RA

RBAC models categorized by Sandhu et al. (2000) are,

- Flat RBAC
- Hierarchical RBAC
- Constrained RBAC
- Symmetrical RBAC

Flat RBAC

In this model, roles are assigned to the users, and permissions are assigned to the roles. Users gain permissions through their assigned roles. Users can have multiple roles. Furthermore, Roles can have numerous permissions too. It is the simplest of all the models. The flat RBAC model, is shown in figure 2.1.

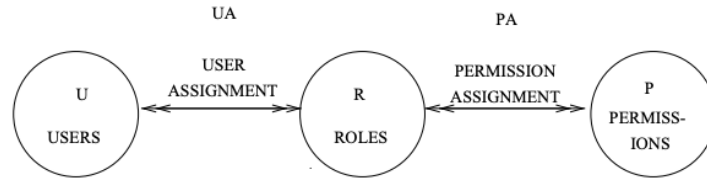


Figure 2.1: Flat RBAC model (Sandhu et al., 2000)

Hierarchical RBAC

The hierarchical RBAC model, is shown in figure 2.2. RH are built on top of the flat RBAC in this model. A role at the top hierarchy level inherits the permissions of all the roles below it. An example of RH is represented in figure 2.3

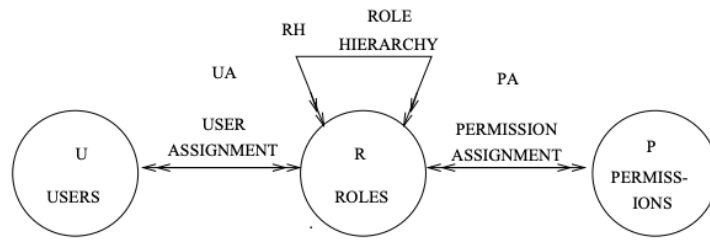


Figure 2.2: Hierarchical RBAC model (Sandhu et al., 2000)

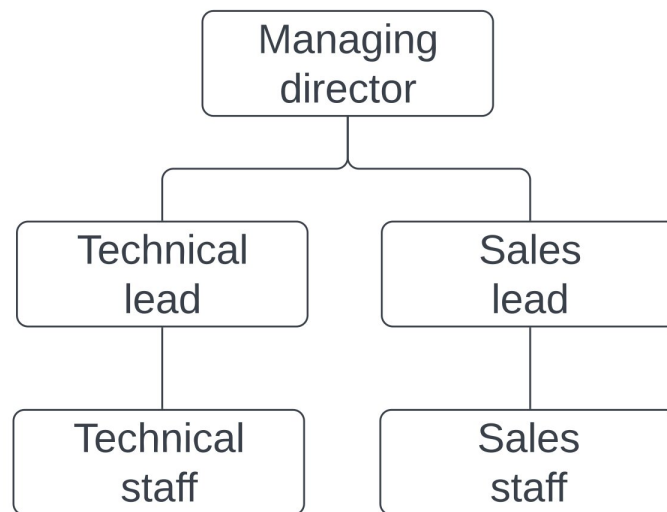


Figure 2.3: Hierarchical role tree

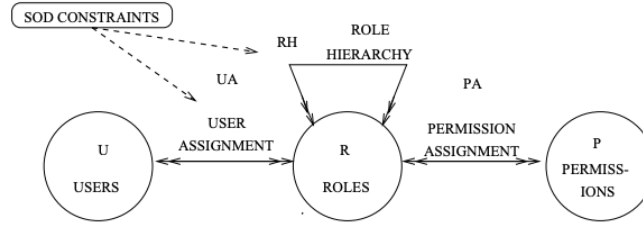


Figure 2.4: Constrained RBAC model with SSoD (Sandhu et al., 2000)

Constrained RBAC

This model imposes constraints on the hierarchical RBAC model with Static Separation of Duties (SSoD) or Dynamic Separation of Duties (DSoD). SSoD ensures that no subject is assigned mutually exclusive roles. As depicted in figure 2.4, the constraints are in RA and RH. For example, a user who is payment initiator cannot be a payment authorizer. Unlike SSoD, where the constraints are predefined, in DSoD, constraints are dynamic and determined at the user session as represented in figure 2.5. DSoD allows the user to have mutually exclusive roles at the same time. For example, a subject can be a payment initiator and payment authorizer, but the subject cannot authorize their own payment.

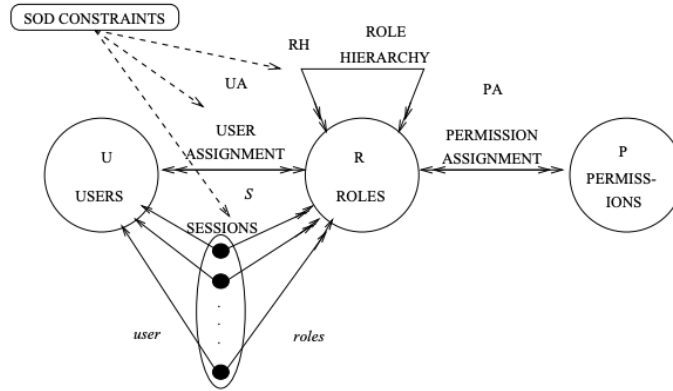


Figure 2.5: Constrained RBAC model with DSoD (Sandhu et al., 2000)

Symmetrical RBAC

Maintaining appropriate permission for respective roles is crucial for authorization. The permission-role review is vital in systems where many administrators can create and assign permissions. As shown in the figure 2.6 and 2.7, this model introduces constraints in PA phase on top of the constrained RBAC model with SSoD and DSoD. It includes an auditing interface for permission-role review.

Based on the review result, obsolete permission can be identified and removed.

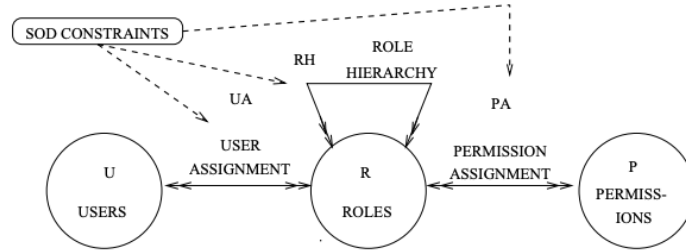


Figure 2.6: Symmetrical RBAC model with SSoD (Sandhu et al., 2000)

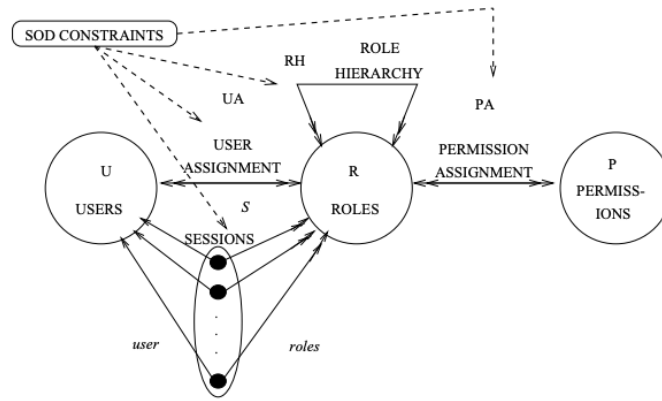


Figure 2.7: Symmetrical RBAC model with DSoD (Sandhu et al., 2000)

3 Requirements

The detailed functional and non-functional requirements to implement PoLP using RBAC is explained in this chapter.

3.1 Functional Requirements (FR)

The following FR are expressed using the FunctionalMASTeR template by Rupp and Sophist. (2014), as shown in figure 3.1. "Condition" field is optional in the template. Keyword "SHALL" indicates the requirement must be fulfilled, "SHOULD" represents it is important to satisfy the requirements but not mandatory for the software to work properly, and "WILL" indicates good to have but not must to have requirements.

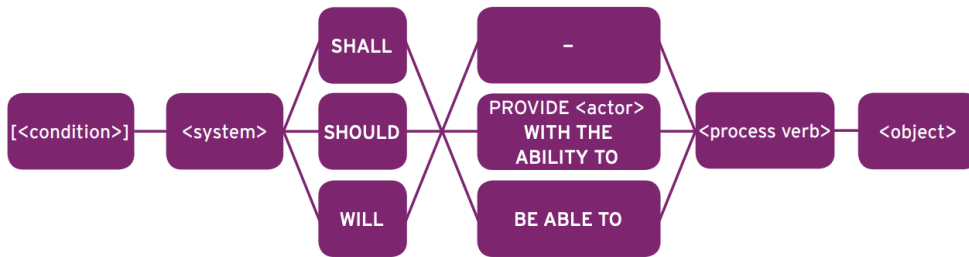


Figure 3.1: FunctionalMASTeR template

The functional requirements of member management and access control are presented in the following subsections.

3.1.1 Member management

FR 1: The QDAcity RBAC system shall be able to define the project permissions listed in table 3.1

FR 2: The QDAcity RBAC system shall be able to define the default project roles Owner, Organizer, Editor and Viewer

Permission	Description
<i>REVISION_CUD</i>	Used to create, update or delete project revision
<i>TODO_CUD</i>	Used to create, update or delete todo
<i>INTER_CODER_AGREEMENT_CUD</i>	Used to create, update or delete project intercoder agreement
<i>CODING_EDITOR_CUD</i>	Used to create, update or delete coding editor related objects
<i>RESOURCES_IMPORT</i>	Used to import codebook
<i>RESOURCES_EXPORT</i>	Used to export codebook and requirements
<i>DESCRIPTION_EDIT</i>	Used to edit project description
<i>SETTINGS_UPDATE</i>	Used to update project settings
<i>MEMBER_CUD</i>	Used to create, update or delete project members
<i>DELETE_PROJECT</i>	Used to delete project

Note: *CUD* is a short form for Create, Update and Delete

Table 3.1: Project permissions and it's description

Permission	Viewer	Editor	Organizer	Owner
Read access to all resources belonging to the project	✓	✓	✓	✓
<i>REVISION_CUD</i>		✓	✓	✓
<i>TODO_CUD</i>		✓	✓	✓
<i>INTER_CODER_AGREEMENT_CUD</i>		✓	✓	✓
<i>CODING_EDITOR_CUD</i>		✓	✓	✓
<i>RESOURCES_EXPORT</i>		✓	✓	✓
<i>RESOURCES_IMPORT</i>			✓	✓
<i>DESCRIPTION_EDIT</i>			✓	✓
<i>SETTINGS_UPDATE</i>			✓	✓
<i>MEMBER_CUD</i>			✓	✓
<i>DELETE_PROJECT</i>				✓

Note: *CUD* is a short form for Create, Update and Delete

Table 3.2: Access control matrix of project roles

FR 3: The QDAcity RBAC system shall be able to assign project permissions to the respective project roles as shown in table 3.2

FR 4: The QDAcity RBAC system shall be able to allow an authorized user to add users with a role in a project

FR 5: The QDAcity RBAC system shall be able to allow an authorized user to add user groups with a role in a project

FR 6: The QDAcity RBAC system shall be able to allow an authorized user to add multiple users and user groups with any supporting role in a project

FR 7: The QDAcity RBAC system shall be able to allow an authorized user to update the role of a project member

FR 8: The QDAcity RBAC system shall be able to allow an authorized user to remove a member from the project

FR 9: The QDAcity RBAC system shall be able to allow an authorized user to leave the project

FR 10: The QDAcity RBAC system shall be able to enforce a constraint that at least one owner member exists in a project

FR 11: The QDAcity RBAC system shall be able to synchronize authorization within the project based on changes with user group/user group users

FR 12: The QDAcity RBAC system should allow the user to define custom roles with permission and assign the custom roles to users/user groups.

FR 13: The QDAcity RBAC system shall be able to aggregate the permissions when a user acquires more than one role in a project

3.1.2 Access control

FR 14: The QDAcity frontend authorization system shall be able to hide or disable non-permitted UI elements

FR 15: The QDAcity backend authorization system shall be able to grant access to an object when the user has permission

FR 16: The QDAcity backend authorization system shall be able to restrict access to an object when the user does not have permission or is unauthorized

3.2 Non Functional Requirements (NFR)

NFR specifies how a system should perform certain functions. It is related to the quality of software. According to the standard ISO-25010¹, software quality can be determined based on the eight metrics as represented in figure 3.2.

The requirements are based on the PropertyMASTeR by Rupp and Sophist. (2014) as represented in figure 3.3. The "condition" and "qualifying expression" fields in the template are optional.



Figure 3.2: ISO/IEC 25010 - Categories

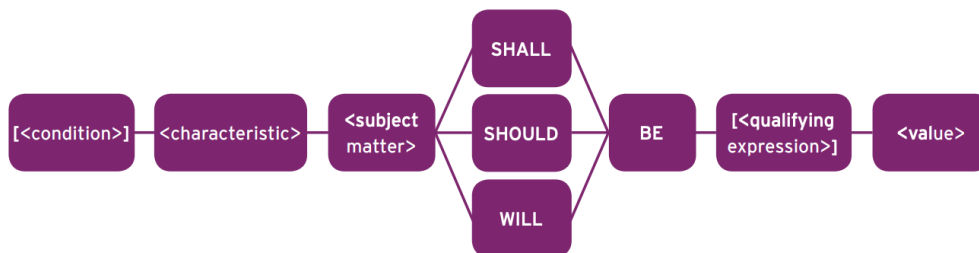


Figure 3.3: PropertyMASTeR template

Functional suitability

NFR 1: The implementation shall be fully functional and should not break any existing functionalities in QDACity

Performance efficiency

NFR 2: The QDACity RBAC system shall be able to cache the default project roles

¹ <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>

Compatibility

NFR 3: The QDAcity RBAC system shall be able to comply with the existing data and data model

NFR 4: The implementation shall be able to work with the existing cloud infrastructure of QDAcity

Usability

NFR 5: The implementation shall be able to support QDAcity's internationalization strategies

NFR 6: The QDAcity RBAC system UI shall be consistent with the current color scheme of QDAcity

Reliability

NFR 7: The QDAcity RBAC system shall behave consistently and throw distinct error codes for different possible error cases

Security

NFR 8: The QDAcity RBAC system Application Programming Interface (API) shall be able to authorize and authenticate the user before performing an action

Maintainability

NFR 9: The implementation shall be able to include unit test cases for the modified functionalities with at least 90% of Lines of Code (LoC) coverage

NFR 10: The implementation shall be able to handle End-to-End (E2E) acceptance test for the modified functionalities

Portability

NFR 11: The implementation shall be able to work as expected in widely used modern browsers such as Google Chrome, Mozilla Firefox and Microsoft Edge

3. Requirements

4 Architecture

This chapter explains the architectural overview of the RBAC system in QDAcity. It is divided into two subsections member management and access control. Both topics have subsections to explain the backend and frontend architecture.

4.1 Member management

QDAcity needs to have a central member management system to control authorization over different objects in QDAcity, like *Project* or *Exercise*.

The system should be integrated easily with minimal effort to any entities that need member management. The following section discusses the backend and frontend architectures of member management.

4.1.1 Backend

There are three different phases in member management. They are

- Member Assignment (MA)
- Role Assignment (RA)
- Permission Assignment (PA)

Figure 4.1 represents the phases in project member management. For better understanding, the phases are discussed in reverse order.



Figure 4.1: Phases in member management of project

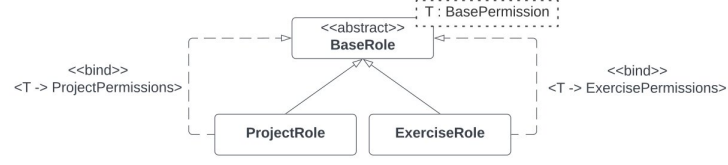


Figure 4.2: UML class diagram of *Role* inheritance

PA

In the PA phase, permissions are assigned to roles. To distinguish between the roles of different entities, we have implemented inheritance from an abstract class which is represented in figure 4.2. The PA part of RBAC is done through role classes. Roles can have one or more permissions. The permission belonging to a role differs based on what type of role it is. For example, the project role will have a specific permission set, while the exercise role will have a different permission set. So the model should be able to make sure that only the correct permissions are assigned to the respective role types. To achieve this, a template parameter T is introduced in abstract *BaseRole* class, which ensures the derived classes extend the base class with its appropriate permission enumeration as T . *ExerciseRole* and *ExercisePermissions* are not implemented in this thesis but used to demonstrate inheritance. For the rest of the thesis, the term *Role* when italicized, will be used to indicate the derived classes of *BaseRole* abstract class, while non italicized form refers to the corresponding linguistic meaning.

RA

In this phase, roles are assigned to the subject. In QDAcity, we refer to subjects with role assignments as members. Similar to *Role* classes, we use inheritance to implement different types of members. The RA part of RBAC is implemented using member classes. The member object contains the role ID assigned to it. A member can only be associated with one role. Roles differ depending on the entity they belong to. For example, a project member can have some roles, while an exercise member can have some different roles. Thus, the model should be able to ensure that only the correct roles are assigned to the respective members. To achieve this, a template parameter T is introduced in the abstract class *BaseMember*, which ensures that the derived classes extend the base class with the appropriate role enumeration as T . The member inheritance is shown in figure 4.3. *ExerciseMember* is not implemented in this thesis but used to demonstrate inheritance. For the rest of the thesis, the term *Member* when italicized, will be used to indicate the derived classes of *BaseMember* abstract class, while non italicized form refers to the corresponding meaning defined for QDAcity.

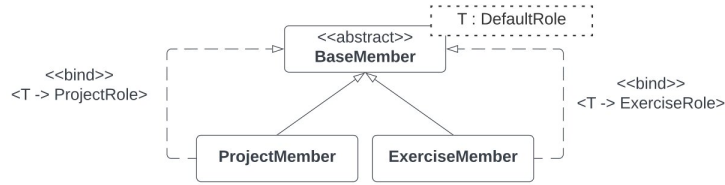


Figure 4.3: UML class diagram of *Member* inheritance

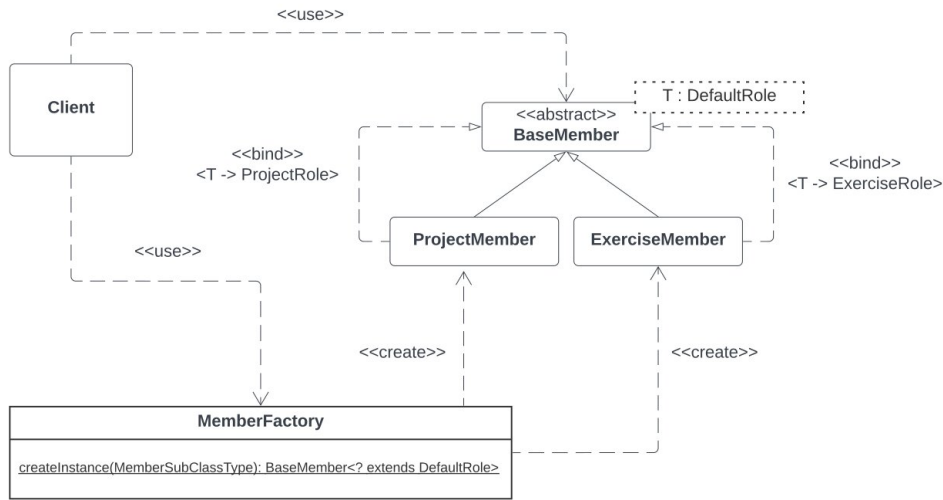


Figure 4.4: *Member* factory UML diagram

A common difficulty with inheritance is creating the instance of different derived classes. We used the factory design pattern to create instances of *Member* classes.

The factory pattern simplifies the creation of derived objects using a simple interface without exposing complex logic. Figure 4.4 represents the factory design pattern for member classes.

MA

In this phase, the members are assigned to the respective entities they belong to. An entity can have multiple members. For example, a project entity can have multiple members with a project role.

The members can be users or user groups. User members are stored in the *directUsers* property, while user group members are stored in the *directUserGroups* property of the associated entity object.

4.1.2 Frontend

QDAcity uses React JS frontend library developed by Facebook.

Member management components in the frontend should be reusable by any entities (*Project* or *Exercise*) which need it. However, the requirements differ for each entity in the following aspects.

- Each entity has its own set of business logic
- Roles and permission vary for different entities
- Certain entities can only allow users to be added
- API differs

So, the components should be decoupled from these constraints to be reusable. The following section discusses how this can be achieved through Separation of Concerns (SoC)

Ingeno (2018) states SoC is a *"design principle that manages complexity by partitioning the software system so that each partition is responsible for a separate concern, minimizing the overlap of concerns as much as possible"*. Abiding by this principle, we can separate UI components from business logic components. The concept is similar to the popular Model View Controller (MVC) pattern, which has bidirectional property binding between components. However, React is not an MVC framework and supports only unidirectional property binding. Benefits of SoC are,

- **Maintainability:** It makes maintenance of a more extensive codebase easier by separating UI components from logic components
- **Reusability:** Different components can reuse UI components with different business logic states
- **Testability:** It will be easy to test the UI components and logic components individually by separating them

Applying SoC to React, the components can be classified¹ as

- Presentation components
- Container components

¹ <https://medium.com/@danabramov/smart-and-dumb-components-7ca2f9a7c7d0>

Presentation components

They have the following characteristics,

- Deals only with the look
- Have Document Object Model (DOM) markups and styles for the look
- Receives data through props
- Emits user actions through the callback provided through props

Some examples are the button and list components, which receive the data as properties from the parent component and emit events to the parent component on user interaction.

Container components

They have the following characteristics,

- Deals with the business logic
- Mostly have state
- Make API calls and mutate the states
- Have minimal DOM elements to wrap the presentation components

Some examples are components containing button and list components that provide data to their children components and make API calls on user interaction such as button click.

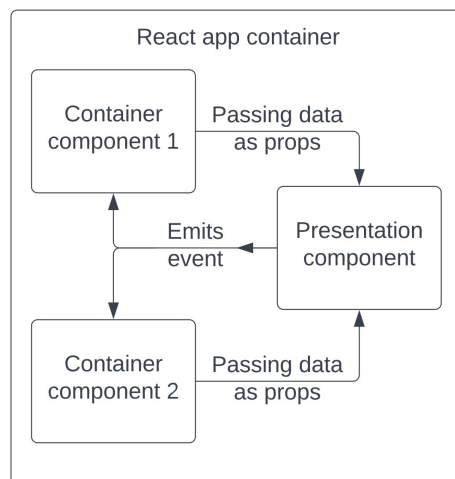


Figure 4.5: Presentation and container components

Figure 4.5 represents the re-usage of presentation components by two different container components in the same app.

4.2 Access control

Access control needs to be implemented in the authorization layer centrally that decides whether a requesting subject should be granted or denied access to do an action on the given object.

There are two ways to check for authorization in an RBAC system, namely explicit access control and implicit access control.

An explicit access control checks for the role, while implicit access control checks for permission.

For example, consider a scenario where a member having an Owner role has *permissionA*. Explicit access control check for this scenario is shown in code 4.1 and implicit access control check is shown in code 4.2.

```
if(member.hasRole(ProjectRole.Owner)){  
    //Do action A  
}
```

Code 4.1: Explicit access control for Owner role

```
if(member.hasPermission(ProjectPermissions.permissionA)){  
    //Do action A  
}
```

Code 4.2: Implicit access control for permission A

Now, if the requirement changes that members with the Editor roles are also allowed to do action A. Then explicit access control needs to be changed as shown in code 4.3.

```
if(member.hasRole(ProjectRole.Owner) ||  
    member.hasRole(ProjectRole.Editor)){  
    //Do action A  
}
```

Code 4.3: Explicit access control for Owner and Editor roles

However, no code refactoring is needed if we use implicit access control. Only *PermissionA* needs to be bound to the Editor role in the PA phase. For better maintainability, we use implicit access control for authorization.

4.2.1 Backend

Authorization is widely used across the code base and is always a crucial part of the system. it should also check if the requesting user is authenticated and performs authorization only if the user is authenticated.

QDAcity already has an authorization layer that acts as a facade hiding the complex implementation and provides a simple interface to the outside world. However, QDAcity does not have authorization checks based on fine granular permission. This thesis has implemented polymorphic methods that handle authorization based on the given permission.

4.2.2 Frontend

In the frontend, access control mostly refers to hiding and disabling of UI elements based on the member's permission on the object. Components across the code-base need Access Permission Data (APD) to hide or disable elements. However, passing APD across the component tree is error-prone. Also, some components within the tree do not even need APD, but they have to pass it to their child, which might need it. The process of passing properties from a parent component to its descendant components at any nested level is called props drilling. In order to avoid the problems with props drilling, react provides a way through context API to send and receive this data without passing it as props at each level by using provider and consumer contexts.

Provider-Consumer pattern

React context API provides a way to share data globally with descendants of the UI component tree. As shown in figure 4.6², data can be provided through the context API at any node of the UI component, and the components below the provider node can consume it. When the data changes, the entire tree under the provider node re-renders.

We can use this pattern to provide APD of the logged-in user in the current entity that can be consumed by the components that need to decide whether it needs to render the UI elements.

² <https://ipraveen.medium.com/react-basic-how-react-16-context-api-work-7257591589fc>

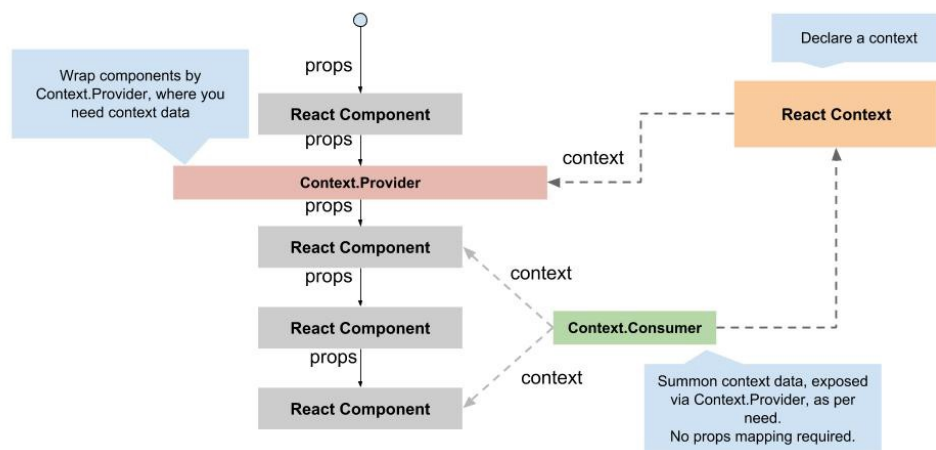


Figure 4.6: Provider - Consumer pattern by React Context

5 Implementation

This chapter discusses the implementation of architectures explained in chapter 4 to satisfy the requirements specified in chapter 3. QDAcity uses Java-based Google App Engine (GAE) for the backend, React for the frontend and Node JS for Real-Time Collaboration System (RTCS). The implementation is explained in two sections member management and access control.

5.1 Member management

In this section, the implementation details of member management are discussed for both the backend and frontend.

5.1.1 Backend

This section describes classes related to RBAC, error handling, additive role model, bit masking techniques and migration of old data to new data model.

Role classes

Role class inheritance is shown in figure 5.1. *BaseRole* is an abstract class, and it implements the serializable interface to deal with the serialization and deserialization for the datastore. It has four properties and one abstract method. Properties of *BaseRole* class are explained in table 5.1. *Role* object contains permissions that are assigned in PA phase. To indicate what type of permission a role can hold, template parameter *T* is introduced in *BaseRole* class.

ProjectRole extends the *BaseRole* class with the *ProjectPermission* enum as parameter *T*. It implements the *hasPermission(ProjectPermissions)* to check whether a project member has permission to do an action on an object. *ProjectPermissions* holds the permission for different actions on different objects belonging to a *Project* entity. Project permissions are listed in table 3.1. As shown in figure 5.2, *ProjectPermission* enum implements *BasePermission* interface to implement common methods. Parameter *T* is constrained with *BasePermission* in *BaseRole*

5. Implementation

class to ensure only the relevant enum that implements this interface alone can be used as permission enum.

A role object holds a value of the type *RoleType* to differentiate between different role types. Role types are

- *DEFAULT_ROLE* - Applicable for system created static roles
- *CUSTOM_ROLE* - Applicable for user created roles
- *DERIVED_ROLE* - Applicable for roles that are combined together from other roles

Roles with *DEFAULT_ROLE* type are static and it should always be available in the Database (DB). To create and persist default role, *CreateRoleServlet* is created and it is triggered on server start. This servlet persist roles, if it is not already available in the DB.

To support Create, Retrieve, Update, Delete (CRUD) operations on role objects, the *RoleController* class is created.

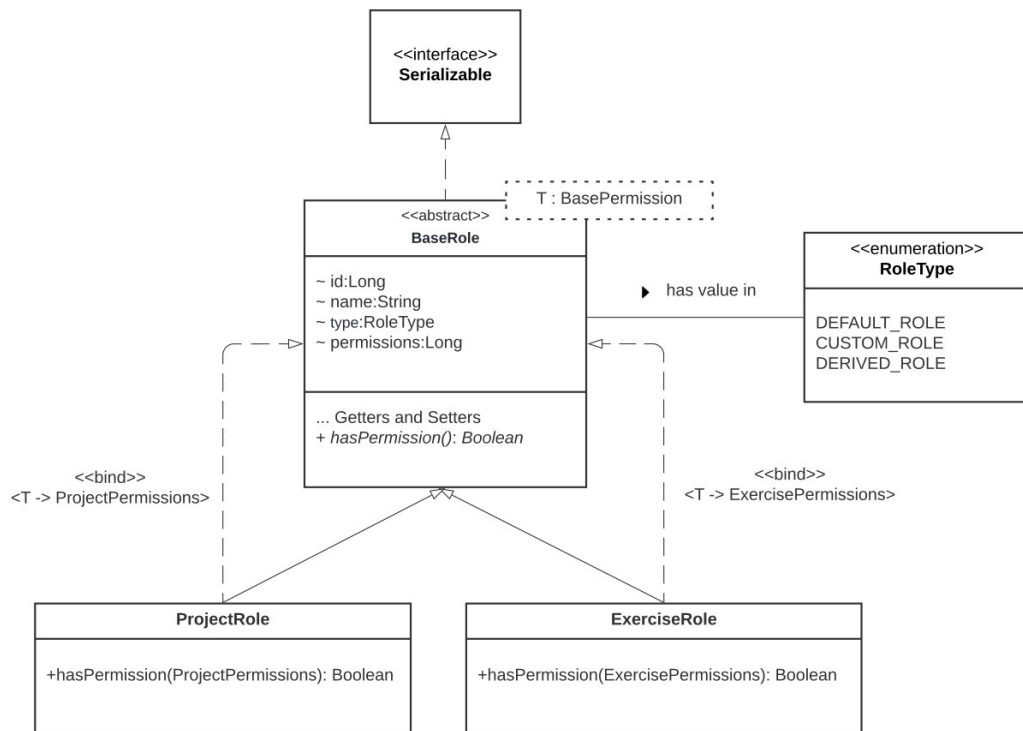
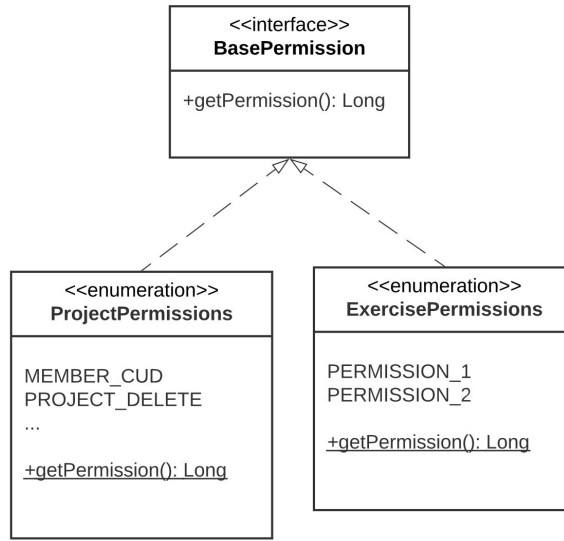


Figure 5.1: UML class diagram of *Role*

**Figure 5.2:** UML class diagram of the permission enums

Property	Data type	Purpose
<i>id</i>	<i>Long</i>	Contains the unique identifier of the object
<i>name</i>	<i>String</i>	Contains the name of the role
<i>type</i>	<i>RoleType</i>	Refers to the enum type, which can be <i>DEFAULT_ROLE</i> / <i>CUSTOM_ROLE</i> / <i>DERIVED_ROLE</i>
<i>permission</i>	<i>Long</i>	Contains the effective permissions of the role, which is obtained by the union of all the permission values assigned to the role in PA phase

Table 5.1: Properties of *BaseRole* class

Member class

Figure 5.3 presents the *Member* inheritance. *BaseMember* is an abstract class, and it implements the serializable interface to deal with the serialization and deserialization for the datastore. *BaseMember* is designed as an abstract class so that it cannot be instantiated and also enforces its derived class to implement specific required methods. It has eight properties and two abstract methods. The properties are explained in table 5.2.

BaseMember class has a template parameter *T* which is constrained with interface *DefaultRole*. Parameter *T* is used in the *hasRole(T)* abstract method,

which checks if a member has a given default role. Derived classes extend this base class with respective role enums as parameter T . For example, *ProjectRole* extends *BaseMember* with *DefaultProjectRole* enum and implements the *hasRole(DefaultProjectRole)*, while *ExerciseRole* extends *BaseMember* with *DefaultExerciseRole* and implements *hasRole(DefaultExerciseRole)*. Parameter T ensures that derived classes implement the *hasRole(T)* method with its respective role enums at compile time. Parameter T is constrained to ensure only relevant enum that implements the *DefaultRole* interface alone can be used so that a random enum cannot be used for implementation. The implementation of this interface by these enums is shown in figure 5.3 .

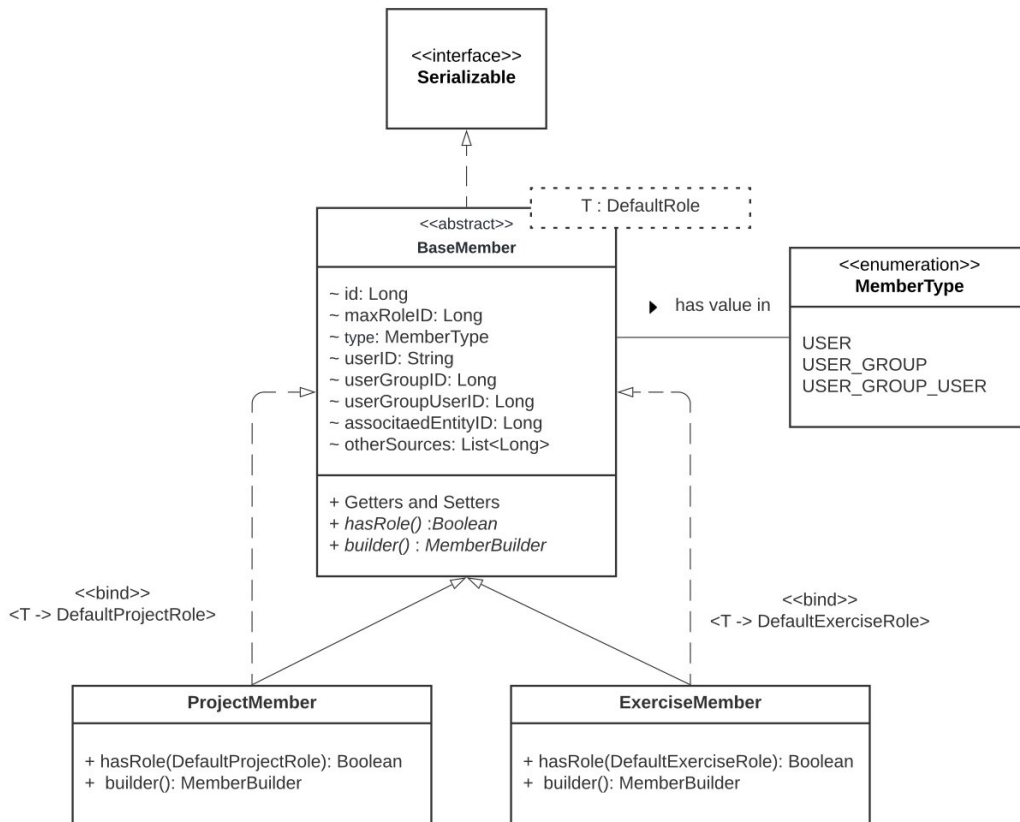


Figure 5.3: UML class diagram of *Member*

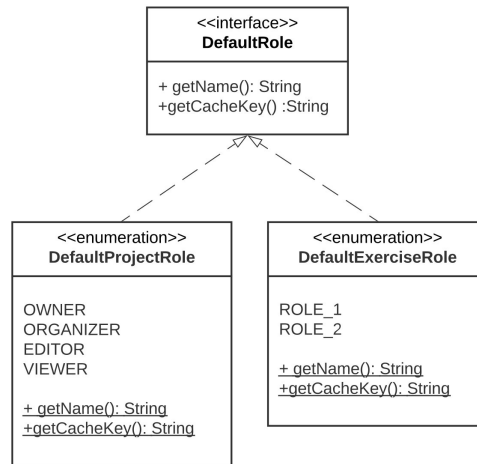


Figure 5.4: UML class diagram of the role enums



Figure 5.5: Project roles hierarchy

A member object holds a value of the type *MemberType* to differentiate between different member types. Different member types are

- *USER* - applicable for users
- *USER_GROUP* - applicable for user groups
- *USER_GROUP_USER* - applicable for users who gain access through user groups

Property	Data type	Purpose
<i>id</i>	<i>Long</i>	Contains the unique identifier of the object
<i>maxRoleID</i>	<i>Long</i>	Contains the role ID of the member with maximum permission
<i>type</i>	<i>MemberType</i>	Refers to enum type which can be <i>USER/USER_GROUP/USER_GROUP_USER</i>
<i>userID</i>	<i>String</i>	Contains the user ID applicable for the types <i>USER</i> and <i>USER_GROUP_USER</i>
<i>userGroupID</i>	<i>Long</i>	Contains the user group ID applicable for the types <i>USER_GROUP</i> and <i>USER_GROUP_USER</i>
<i>userGroupUserID</i>	<i>Long</i>	Contains the user group user ID applicable only for the type <i>USER_GROUP_USER</i>
<i>associatedEntityID</i>	<i>Long</i>	Contains the ID of the entity (<i>Project/Exercise/UserGroup</i>) to which the member is part of.
<i>otherSources</i>	<i>List<Long></i>	Contains the list of <i>BaseMember</i> ID in which the user has same maximum role through some other <i>USER_GROUP</i> member.

Table 5.2: Properties of *BaseMember* class

As discussed above, the *ProjectMember* class has only two methods and no properties. Therefore, it might not seem to be a good candidate for inheritance. However, having inheritance within our initial design helps extend the functionalities for future customization. For example, consider that only users are allowed and user groups are not allowed to be added to the *Exercise* entity. Then with the chosen design, we can override the getters and setters related to the user group in the derived class to persist *null*. This way, we can ensure that no user groups are added even if the methods are invoked in the code.

As depicted in figure 5.4, *DefaultProjectRoles* enum has the roles owner, organizer, editor and viewer. This enum implements the *DefaultRole* interface. Implementation of *DefaultProjectRoles* is shown in the figure 5.5.

To support CRUD operations on member objects, the *MemberController* class is created.

Based on the *MemberType*, many properties need to be set. The builder pattern helps construct complex objects step by step. Using the same construction code, different representations of objects can be created. A UML diagram for the builder pattern to create different *Member* objects initialized with different properties are presented in figure 5.6.

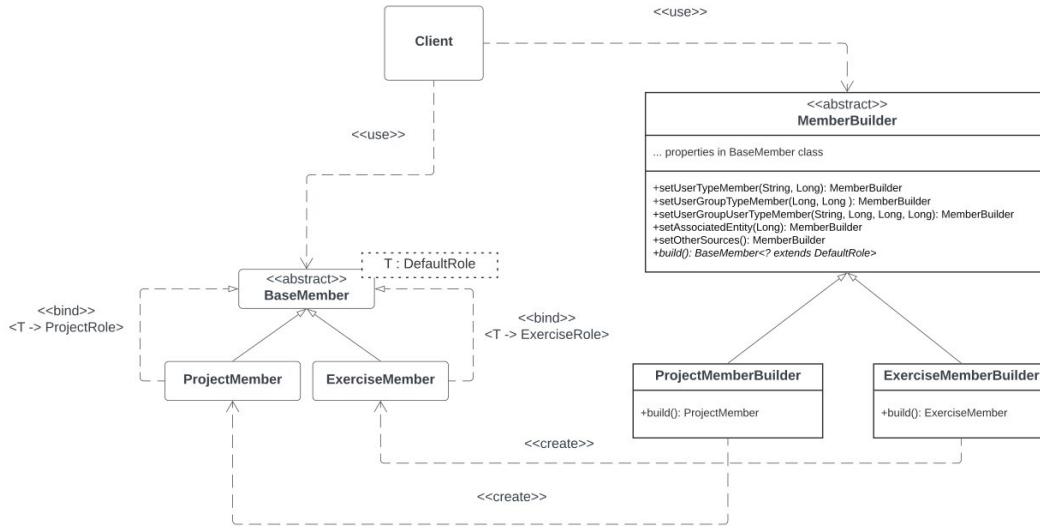


Figure 5.6: *Member* builder UML diagram

Project class

A project should allow adding users and user groups, and a project should be able to differentiate whether a member is a user or user group. As described in section 4.1.1, the *directUsers* and *directUserGroups* properties were added in the Project class. The prefix *direct* indicates that these members were added directly to the project. There are also indirect members who gain access through user groups. To differentiate between these types of members *MemberType* enum with values of *USER*, *USER_GROUP*, *USER_GROUP_USER* is introduced.

In addition to these properties, two more properties, *maxRoleMembers* and *supportedRoles*, were also added. These properties are explained in table 5.3.

Property	Data type	Purpose
<i>directUsers</i>	<i>List<Long></i>	Contains the list of <i>ProjectMember</i> IDs with <i>USER MemberType</i>
<i>directUserGroups</i>	<i>List<Long></i>	Contains the list of <i>ProjectMember</i> IDs with <i>USER_GROUP MemberType</i>
<i>maxRoleMembers</i>	<i>List<Long></i>	Contains all the <i>ProjectMember</i> Ids of distinct direct or indirect users with their maximum role.
<i>supportedRoles</i>	<i>List<Long></i>	Contains the list of <i>ProjectRole</i> Ids of all the supporting roles, including default project roles and custom roles, if any.

Table 5.3: RBAC properties of *Project* class

JDO persistence strategies

We use the *SUPER_CLASS_TABLE* inheritance strategy provided by Java Data Objects (JDO) for both *Member* and *Role* classes. This strategy instructs the database management system to store all the types of *BaseMember* and *BaseRole* classes in one table each. Since we are building a central RBAC system for QDAcity, it is better to use super class inheritance. According to the JDO guide¹, this has the benefit that retrieving an object is a single call to a single table. However, it also has the drawback that the single table can have many columns, which suffer database performance and readability. Thus a discriminator column is desired.

Based on the JDO mapping guide², a discriminator is an additional column stored alongside the data to recognize the class to which that information belongs. It is useful to determine the object type upon retrieval. Two types of discriminators supported by JDO are,

- class-name - the class name is stored as a discriminator
- value-map - unique numeric value for each type is stored as a discriminator

We use the class-name discriminator strategy to identify different object types.

¹ https://www.datanucleus.com/products/accessplatform_4_1/jdo/orm/inheritance.html

² https://www.datanucleus.org/products/accessplatform_5_1/jdo/mapping.html

Error handling

Error handling in API is very important as it helps to convey the client what has caused the error. In the RBAC system, to handle different kinds of error cases, *RBACErrorCode* enum is created with the following items,

- *MEMBER_NOT_AVAILABLE* - used when the given member is not found in datastore
- *ROLE_NOT_SUPPORTED* - used when the given role is not supported by the project
- *ROLE_NOT_CHANGED* - used when the role is not changed while updating member role
- *SINGLE_OWNER_CASE* - used when the given member is the only owner of the project while degrading role/removing member/member leaves the project
- *MEMBER_ALREADY_ADDED* - used when the member is already added to the project
- *USER_GROUP_USER_DIRECT_ACTION_NOT_ALLOWED* - used when an action is directly performed on a user group user
- *USER_CANNOT_REMOVE_THYSELF* - used when the member remove themselves
- *OWNER_MEMBER_NOT_FOUND* - used when no owner member found for the project
- *INVALID_EMAIL* - used when the user email is invalid

MemberController has methods to check and throw *RBACException* for each error case. API methods catch these exceptions and throw them as *BadRequestException* with the specific error code to the client.

Nomenclature of permission

Permissions are the core part of RBAC. The permission name should express what action can be done on which object. So a common nomenclature is followed.

OBJECTNAME_ACTIONS

The actions are sometimes grouped as Create, Update, Delete (CUD), which refers to the short form of Create, Update and Delete. The read property of the commonly known abbreviation CRUD is handled separately to allow read access to non-registered users under certain conditions. If these conditions apply, those users could not be authorized to use RBAC because they can not be

authenticated.

Additive role model

QDAcity currently has hierarchical default roles. So when a user is assigned multiple roles, the role with maximum permission will become the effective permission for that user. However, when custom roles are introduced, the roles may not be hierarchical, and the maximum role cannot be determined. As depicted in figure 5.7, when multiple roles are assigned to an user within the same entity, the user's effective permission will be a union of all the permission sets of the user's assigned roles. This is called an additive role model.

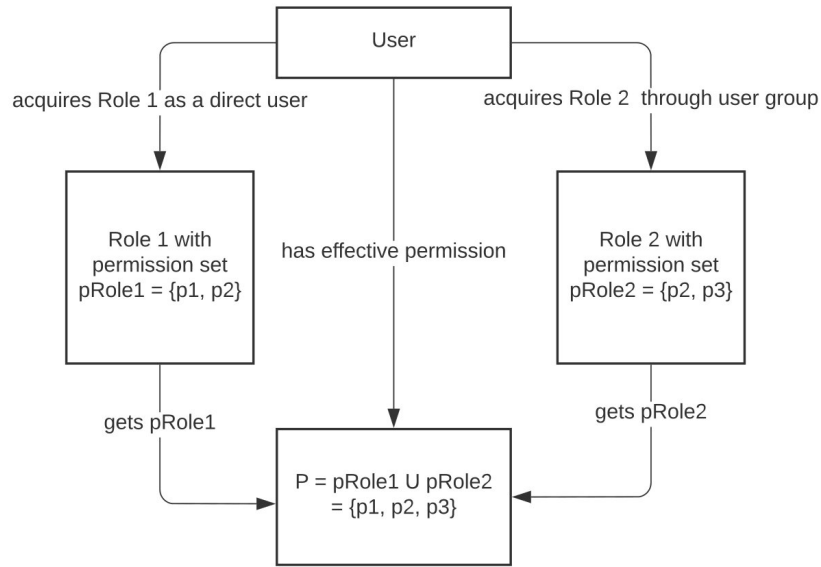


Figure 5.7: Additive role model

Bit masking technique

As the app matures, functionality will increase. Eventually, the number of possible permissions may also grow. Then it will not be easy to store different permission boolean in the DB. To solve these difficulties, the bit masking technique is used.

A bitmask is a sequence of bits used to represent multiple boolean flags. If a bit is set, it is considered *true*, otherwise *false*. Based on this concept, each permission is assigned a unique value in the powers of two. Aggregating all the appropriate permissions of a role results in its effective permission. A mask is a filter that extracts needed bits and leaves the unwanted bits. Masking is

the process of applying mask to the value. Bitwise operations used for different masking purposes are

- Bitwise AND - used to extract needed bits
- Bitwise OR - used to add bits
- Bitwise XOR - used to remove the bits

Let us consider the permissions with values in the sequence of powers of 2 as shown in table 5.4 and access control matrix on table 5.5. The blue represents the bitmask value of the permission and orange color represents the effective permission of the role.

Permission	Value	Bitmask value
<i>TODO_CUD</i>	$2^0 = 1$	0001
<i>VALIDATION_PROJECT_APPROVE</i>	$2^1 = 2$	0010
<i>PROJECT_DELETE</i>	$2^2 = 4$	0100

Table 5.4: Permission and bitmask value

Permission	Editor	Organizer
<i>TODO_CUD</i>	✓	✓
<i>VALIDATION_PROJECT_APPROVE</i>		✓
<i>PROJECT_DELETE</i>		
Effective permission	$2^0 = 1$ = 0001	$2^0 + 2^1 = 3$ = 0011

Table 5.5: Access control matrix roles with effective permission

Check permission

Logical AND(&) can be used to mask the given permission from the role's effective permission. If the masked value is greater than zero, permission is granted else denied.

The formula to check permission is,

$$bitMaskValueOfPermissionToCheck \& effectivePermission > 0 \quad (5.1)$$

Checking the permission *TODO_CUD* for the editor role, the condition evaluates to be *true* (0001 & 0001 = 0001 > 0), so editor member has the permission *TODO_CUD*

Checking the permission *PROJECT_DELETE* for the editor role, the condition evaluates to be *false* (0001 & 0100 = 0000 $\not>$ 0), so editor member has no permission *PROJECT_DELETE*.

Add permission

Logical OR(|) can be used to add a given permission to the existing effective permission.

The formula to add a permission is,

$$effectivePermission = bitMaskValueOfPermission | effectivePermission \quad (5.2)$$

Adding `VALIDATION_PROJECT_APPROVE` permission to the editor role, effective permission will become `0010 | 0001 = 0011`

Remove permission

Logical XOR(^) can be used to remove a given permission from the existing effective permission.

The formula to remove permission is,

$$effectivePermission = bitMaskValueOfPermissionToRemove ^ effectivePermission \quad (5.3)$$

Removing `VALIDATION_PROJECT_APPROVE` permission from the organizer role, the effective permission will become `0010 ^ 0011 = 0001`

Migration of data

Previously, project users were stored in a list called *owners* in the project entity. With the current RBAC model, all the user data are stored in the *Member* table. To migrate project user data to the new RBAC model, we used task queues. Figure 5.8 depicts the workflow of migration. When a client triggers the *updateProjectRBACProperties* API, deferred task *UpdateAllProjectsRBACEntries* is added to the queue. This task queries all the projects from the datastore and adds *UpdateSingleProjectRBACEntries* deferred task for every project to the queue. Queues are consumed by the worker service, which calls the deferred task handlers. Then the handlers create a *ProjectMember* object with *USER MemberType* and *maxRoleId* as owner for each users and persist them in the datastore.

Synchronization of user group with project members

Whenever there is a change in a user group, project members needs to be synchronized with the user group. A deferred task is created to compute and persist the members and their maximum permission.

This task will be added to the queue when,

- the user group is deleted
- user leaves the user group
- user is added to the user group
- user is removed from the user group

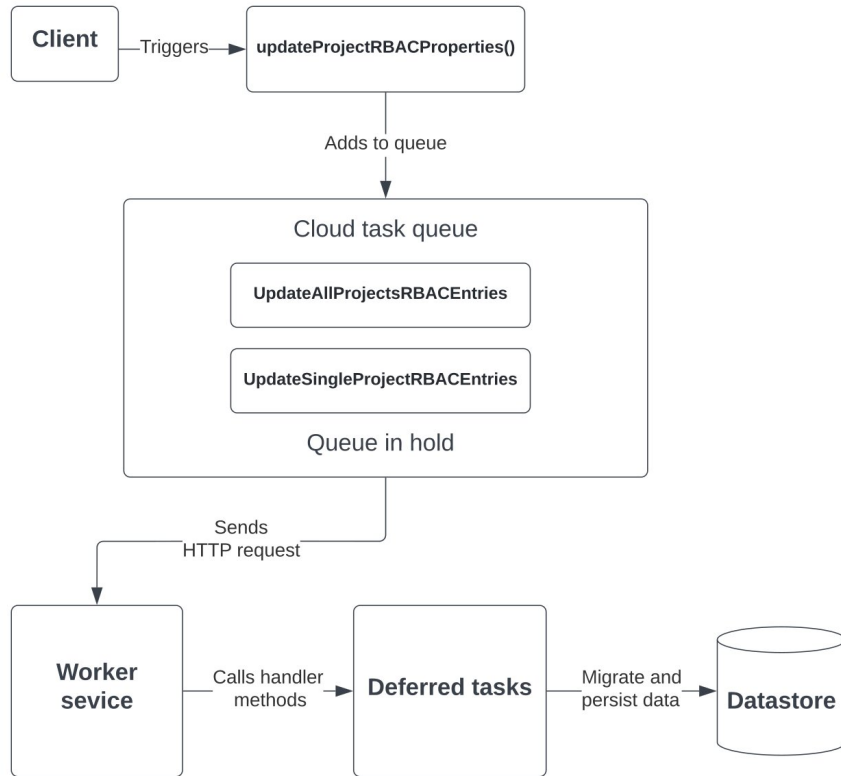


Figure 5.8: Migration workflow of data to RBAC model

5.1.2 Frontend

In the frontend, we use container and presentation components to implement the SoC design principle. The container component *ProjectMembers* handles the network requests and provide the data as properties to the presentation components. On user interaction, presentation components emit events to the *ProjectMembers* component and react to property changes on state update of container component.

The components hierarchy is represented in figure 5.9.

5. Implementation

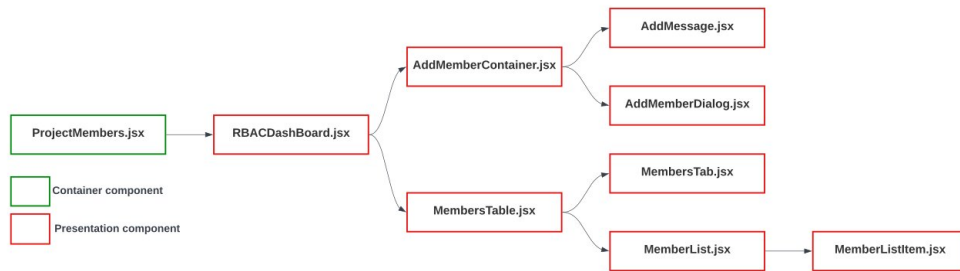


Figure 5.9: Components hierarchy of container and presentation components

5.2 Access control

5.2.1 Backend

In this section, the implementation of access control mechanisms for backend and frontend are discussed.

Backend

As discussed, we use implicit access control in the *Authorization* facade. Authorization involves many steps before deciding whether to deny or allow the request. The general workflow of authorization in the backend is represented as a flow chart in figure 5.10.

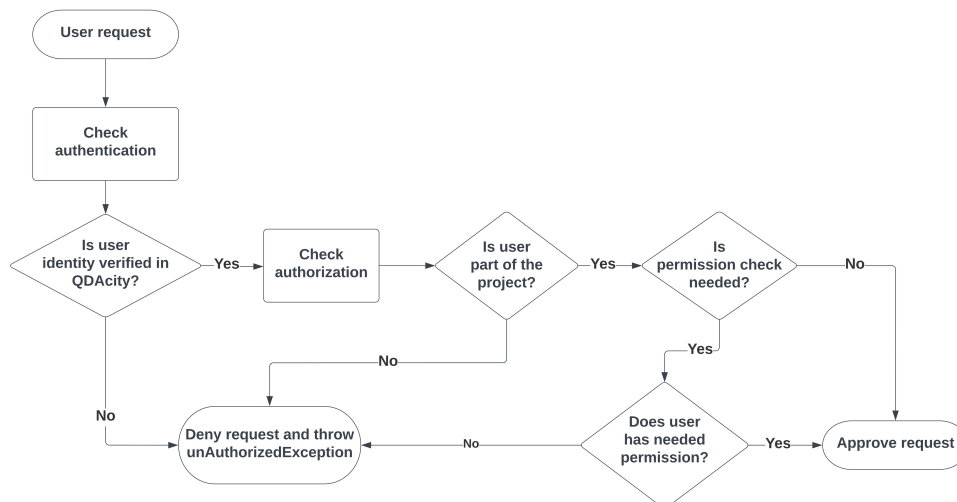


Figure 5.10: Authorization flowchart for RBAC

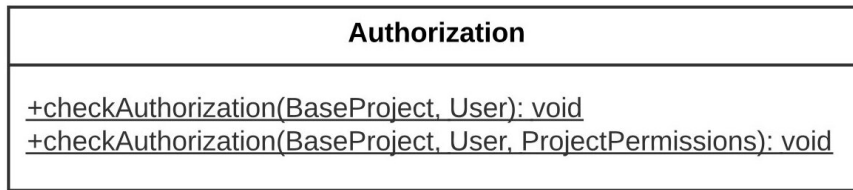


Figure 5.11: Polymorphic authorization methods

As depicted in the flow chart, authorization involves lot of checks. The logic also varies based on the object of interest. To implement implicit access control polymorphic methods were created already in *Authorization* facade. Permission-based authorization methods were created and overloaded with other methods based on the different object types, types of the user object, and with or without permission. A UML class diagram of polymorphic methods with and without permission is represented in figure 5.11. For GET http requests, only existence in project is checked while for other requests permission is also checked.

5.2.2 Frontend

In the frontend, access control should show/hide the UI elements based on the underlying permission.

Provider consumer pattern

We use the provider consumer of the context API to pass the access permission data to different components. Wherever a permission check is needed, the permission consumer should be used to get the permission and conditionally render UI elements based on particular permission. The following code handles access control to show or hide "Create revision" button using this pattern.

```

<PermissionProvider permission={permissions}>
  ...
  <PermissionConsumer>
    {permission =>
      {permission.REVISION_CUD &&
        <button>Create revision</button>
      }
    }
  </PermissionConsumer>
  ...
</PermissionProvider>

```

Code 5.1: Access control in frontend without containment component

Containment component

The problem with using provider consumer alone is that there is a lot of boilerplate code with conditional rendering to show or hide UI elements based on the permission. Also, the permission consumer has to be imported in all the components which need access control. To avoid these difficulties, containment components are used. React provides children properties to components in the name *child*. This will be helpful when the component does not know the children in advance, but needs to wrap it. Based on this concept, the *ShowIfPermitted* containment component is created, which accesses the permission from *PermissionConsumer*, receives the permission to be checked, and children as properties. If the given permission is satisfied, the children components are returned and rendered. Similarly, *DisableIfNotPermitted* containment component is also created to disable UI elements when not permitted. The following code handles access control to show or hide "Create revision" button using containment component.

```
<PermissionProvider permission={permissions}>
  ...
  <ShowIfPermitted
    permission={ProjectPermissions.REVISION_CUD}>
    <button>Create revision</button>
  </ShowIfPermitted>
  ...
</PermissionProvider>
```

Code 5.2: Access control in frontend with containment component

6 Evaluation

In chapter 3, we presented the FR and NFR. In this chapter, we examine whether these requirements are satisfied or not.

6.1 Functional Requirements

This section evaluates FR for member management and access control.

6.1.1 Member management

FR 1: The QDAcity RBAC system shall be able to define the project permissions listed in table 3.1

As discussed in section 5.1.1, ten project permissions were defined in an enum *ProjectPermission*.

We satisfied this requirement.

FR 2: The QDAcity RBAC system shall be able to define the default project roles Owner, Organizer, Editor and Viewer

As described in section 5.1.1, the four project roles were defined in an enum *ProjectRole*.

We satisfied this requirement.

FR 3: The QDAcity RBAC system shall be able to assign project permissions to the respective project roles as shown in table 3.2

The project permissions were assigned to the project roles in the PA phase of RBAC. The permissions were added together to form the effective permission of a role as mentioned in section 5.1.1.

We satisfied this requirement.

FR 4: The QDAcity RBAC system shall be able to allow an authorized user to add users with a role in a project

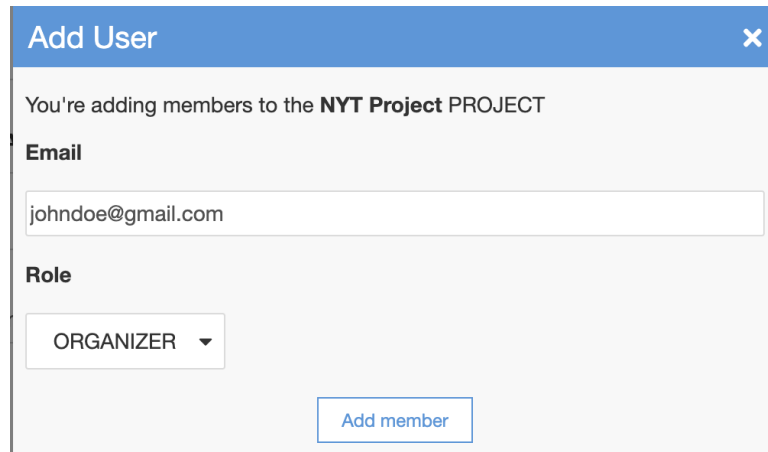
A modal dialog box titled "Add User" with a close button (X) in the top right corner. The dialog contains a message: "You're adding members to the **NYT Project** PROJECT". Below this, there are two sections: "Email" and "Role". The "Email" section has a text input field containing "johndoe@gmail.com". The "Role" section has a dropdown menu currently showing "ORGANIZER". At the bottom right of the dialog is a blue button labeled "Add member".

Figure 6.1: Add User dialog

AddUser API was created in *ProjectEndpoint* to support user addition to a project. In the frontend, a modal dialog was implemented to enable user addition by entering an email and choosing a role. Figure 6.1 shows the add user dialog in QDAcity.

We satisfied this requirement.

FR 5: The QDAcity RBAC system shall be able to allow an authorized user to add user groups with a role in a project

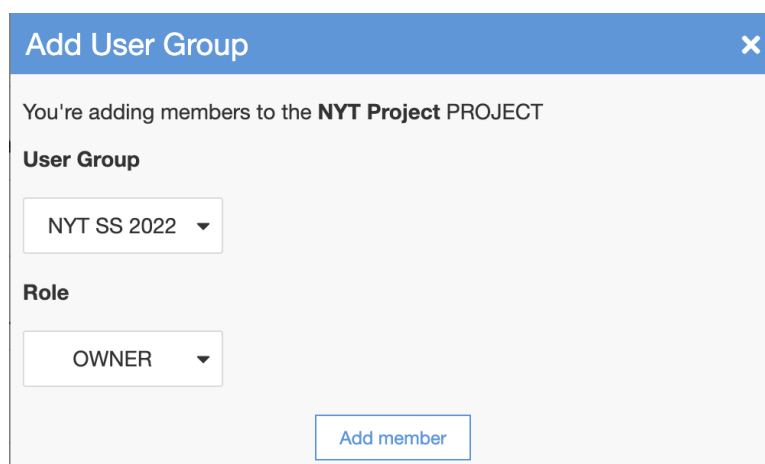
AddUserGroup API was created in *ProjectEndpoint* to support user group addition to a project. A modal dialog was implemented in the frontend to enable user group addition by choosing an owned user group and a role. Figure 6.2 shows the add user group dialog in QDAcity.

We satisfied this requirement.

FR 6: The QDAcity RBAC system shall be able to allow an authorized user to add multiple users and user groups with any supported role in a project

Both *AddUser* and *AddUserGroup* APIs allow multiple distinct members to be added to the project. The listing of multiple members is shown in figure 6.3

We satisfied this requirement.



Add User Group [X]

You're adding members to the **NYT Project** PROJECT

User Group

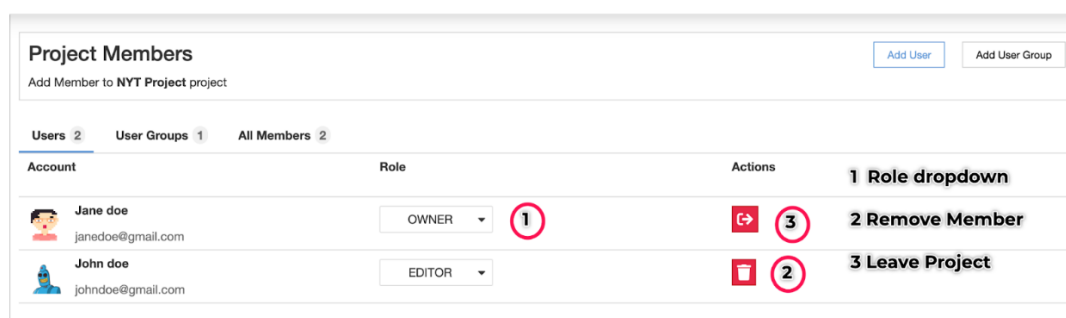
NYT SS 2022 ▼

Role

OWNER ▼

Add member

Figure 6.2: Add user group dialog



Project Members [Add User] [Add User Group]

Add Member to NYT Project project

Users 2 | User Groups 1 | All Members 2






Account	Role	Actions
 Jane doe janedoe@gmail.com	OWNER ▼ 1	 3 1 Role dropdown  2 2 Remove Member
 John doe johndoe@gmail.com	EDITOR ▼	 2 3 Leave Project

Figure 6.3: RBAC project members list UI

FR 7: The QDAcity RBAC system shall be able to allow an authorized user to update the role of a project member

updateMemberRole API was added in *ProjectEndpoint* to modify a member role. A dropdown with supported roles was provided in the frontend for the authorized user to change the member role. The roles dropdown is encircled as 1 in figure 6.3

We satisfied this requirement.

FR 8: The QDAcity RBAC system shall be able to allow an authorized user to remove a member from the project

removeMember API was added in *ProjectEndpoint*. In the frontend, a button to remove a user, user group members were provided. The remove member button is encircled as 2 in figure 6.3

We satisfied this requirement.

FR 9: The QDAcity RBAC system shall be able to allow an authorized user to leave the project

leaveProject API was added in *ProjectEndpoint*. In the frontend, a button to leave the project was added only for logged-in users. The leave project button is encircled as 3 in figure 6.3

We satisfied this requirement.

FR 10: The QDAcity RBAC system shall be able to enforce a constraint that at least one owner member exists in a project

The *throwIfMemberIsOnlyOwner()* method has been implemented in *MemberController* to check if the specified member is the only owner of the project and throws *RBACException* if a member is the only owner. This method was used in *updateMemberRole*, *removeMember*, and *leaveProject* APIs to ensure that at least one owner member exists for the project. In addition to the API error handling, error message prompt to the user was also handled in the frontend.

We satisfied this requirement.

FR 11: The QDAcity RBAC system shall be able to synchronize authorization within the project based on changes with user group/user group users

As discussed in section 5.1.1, whenever there is a change in the user group/user group users, deferred tasks are added to the queue to synchronize project authorization with the changed user group.

We satisfied this requirement.

FR 12: The QDAcity RBAC system should allow the user to define custom roles with permission and assign the custom roles to users/user groups

Due to time constraints and lower priority, the custom role functionality was not implemented. Nevertheless, the RBAC system's design and architecture were implemented to be extendable for custom roles.

We did not satisfy this requirement.

FR 13: The QDAcity RBAC system shall be able to aggregate the permissions when a user acquires more than one role in a project

As discussed in section 5.1.1, RBAC was implemented with additive role model which aggregates the role's permission when a user has multiple roles in a project.

We satisfied this requirement.

6.1.2 Access control

FR 14: The QDAcity frontend authorization system shall be able to hide or disable non-permitted UI elements

As discussed in section 5.2.2, we have implemented the provider-consumer pattern along with *ShowIfPermitted* component to show/hide UI elements and *DisableIfNotPermitted* component to disable UI elements.

We satisfied this requirement.

FR 15: The QDAcity backend authorization system shall be able to grant access to an object when the user has permission

As discussed in section 5.2.1, the authorization layer has been implemented to check authentication and authorization based on permission. Requests will be allowed access for the object if the member has the needed permission.

We satisfied this requirement.

FR 16: The QDAcity backend authorization system shall be able to restrict access to an object when the user does not have permission or is unauthorized

As discussed in section 5.2.1, the authorization layer has been implemented to check authentication and authorization based on permission. If authorization fails, request will be denied access by throwing *UnauthorizedException*.

We satisfied this requirement.

6.2 Non Functional Requirements

In this section, we evaluate the NFR listed in chapter 3.

Functional suitability

NFR 1: The implementation shall be fully functional and should not break any existing functionalities in QDAcity

Manual test was done to confirm that the functionalities were working as expected. All the unit test cases were also passed, indicating all the existing functionalities work as expected.

We satisfied this requirement.

Performance efficiency

NFR 2: The QDAcity RBAC system shall be able to cache the default project roles

All the project roles were stored in *Memcache* (in-memory key-value storage) with both id and name as keys to facilitate retrieval of roles for different purposes.

We satisfied this requirement.

Compatibility

NFR 3: The QDAcity RBAC system shall be able to comply with the existing data and data model

Deferred tasks were implemented to migrate the project user data from the old data model to the new RBAC data model successfully.

We satisfied this requirement.

NFR 4: The implementation shall be able to work with the existing cloud infrastructure of QDAcity

The RBAC system was built using the existing cloud infrastructures of QDAcity (GAE, datastore, *Memcache* and tasks queue) without any additional infrastructure.

We satisfied this requirement.

Usability

NFR 5: The implementation shall be able to support QDAcity's internationalization strategies

All the messages used in the UI were internationalized in English and German using QDAcity's internationalization system. The Continuous Integration (CI) pipeline task to check translation messages also confirms it.

We satisfied this requirement.

NFR 6: The QDAcity RBAC system UI shall be consistent with the current color scheme of QDAcity

The new RBAC system in the frontend was built using many common components and newly added components only use colors defined in the theme file.

We satisfied this requirement.

Reliability

NFR 7: The QDAcity RBAC system shall behave consistently and throw distinct error codes for different possible error cases

All unit test cases related to the RBAC system pass consistently, indicating that the functionalities behave as expected. It also includes tests that check for the error cases. As discussed in section 5.1.1, *RBACErrorCode* enum and

RBACException was created in the backend to throw exception with different error codes for different error cases.

We satisfied this requirement.

Security

NFR 8: The QDAcity RBAC system API shall be able to authorize and authenticate the user before performing an action

Authentication and authorization of all the RBAC APIs were done using common methods in the *Authorization* facade. These API throws *UnauthorizedException* exception whenever authentication or authorization is failed.

We satisfied this requirement.

Maintainability

NFR 9: The implementation shall be able to include unit test cases for the modified functionalities with at least 90% of LoC coverage

We have written 35 unit test cases related to RBAC system in the backend. LoC coverage percentage for the RBAC packages are listed in table 6.1. The table shows that only two packages meet 90% criteria while others don't.

We partially satisfied this requirement.

Package	Lines missed	Lines covered	LoC coverage in %
<i>com.qdacity.rbac</i>	0	12	100
<i>com.qdacity.rbac.member</i>	21	93	83.03
<i>com.qdacity.rbac.member.controller</i>	80	343	81.09
<i>com.qdacity.rbac.role</i>	16	37	69.81
<i>com.qdacity.rbac.role.controller</i>	32	63	66.32
<i>com.qdacity.rbac.deferredTasks</i>	39	43	52.44
<i>com.qdacity.project.rbac.member</i>	2	10	83.33
<i>com.qdacity.project.rbac.role</i>	22	99	81.81
<i>com.qdacity.endpoint</i> (includes only RBAC related APIs in <i>ProjectEndpoint</i> class)	0	124	100

Table 6.1: LoC coverage for unit test

NFR 10: The implementation shall be able to handle E2E acceptance test for the modified functionalities

Due to time constraints and lower priority, we could not able to implement E2E acceptance test.

We did not satisfy this requirement.

Portability

NFR 11: The implementation shall be able to work as expected in widely used modern browsers such as Google Chrome, Mozilla Firefox and Microsoft Edge

Manual testing ensured that the functionalities worked as intended in the recent version of these browsers.

We satisfied this requirement.

7 Future Work

Although QDAcity now has a central RBAC system, there is still room for improvements. This section describes future works to be done in QDAcity.

7.1 Extending RBAC to other entities

Currently, RBAC is implemented only for project entities. However, QDAcity has other entities like *UserGroup*, *Course*, and *Exercise* where RBAC is a good candidate. Therefore, going forward, the RBAC system should be integrated with these entities as additional RBAC objects.

7.2 Custom roles

Custom roles are user-defined roles. The user should assign permission bindings when creating the role. RBAC system has been designed and implemented to support custom roles. In the future, custom roles should be implemented as an option for users.

7.3 RTCS integration

When a member is added, modified role, or deleted from a project, it is not communicated to the other members collaborating on the same project in real-time. This leads to inconsistent behavior between the frontend and backend, which are only eventually resolved on reload of the frontend client. By integrating with RTCS, inconsistencies will be eliminated, providing a smooth user experience.

7. Future Work

8 Conclusion

The objective of this project was to implement PoLP in QDAcity to control collaboration.

To successfully implement PoLP, we started by surveying literature related to information security and access control, as discussed in chapter 2. We found that RBAC is the most appropriate model to achieve PoLP. In chapter 3, we stated FR and NFR related to member management and access control. In chapter 4, we proposed an architecture to build these functionalities. The proposed architectures were implemented as discussed in chapter 5. In chapter 6, we evaluated all the 27 requirements, out of which we have satisfied 24 requirements fully satisfied, 1 partially satisfied, and 2 not satisfied. In chapter 7, improvements needed to make QDAcity member management better has been discussed.

QDAcity now has central member management based on the RBAC concept. It allows users and user groups to be added to the project with roles. Member's roles can be updated, and members can be removed. Users are granted or denied access based on their underlying role's permission.

The resulting RBAC system was built using widely used design patterns. Inheritance class hierarchies were used to support different types of members and roles. Most of the queries related to the RBAC system were done using low-level data-store API for better performance. In the frontend, SoC and provider-consumer pattern were used. Also, the migration of existing data to the new RBAC data model was done smoothly.

To conclude, hierarchical RBAC system with four project roles (Owner, Organizer, Editor and Viewer) and ten fine-granular permissions were successfully implemented to support PoLP and control the collaboration in QDAcity.

8. Conclusion

References

- Edwards, W. K. (1996). Policies and roles in collaborative applications. *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, 11–20. <https://doi.org/10.1145/240080.240175>
- Ferraiolo, D., Kuhn, D. & Chandramouli, R. (2003). *Role-based access control*. Artech House.
- Ingeno, J. (2018). *Software architect's handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing.
- Mezmir, E. A. (2020). Qualitative data analysis: An overview of data reduction, data display and interpretation. *Research on Humanities and Social Sciences*.
- Rupp, C. & Sophist. (2014). *Requirements - engineering und - management – pro- fessionelle, aus der praxis von klassisch bis agil*. Carl Hanser Verlag München.
- Sandhu, R., Ferraiolo, D. & Kuhn, R. (2000). The nist model for role-based access control: Towards a unified standard. *Proceedings of the Fifth ACM Workshop on Role-Based Access Control*, 47–63. <https://doi.org/10.1145/344287.344301>
- Sandhu, R., Coyne, E., Feinstein, H. & Youman, C. (1996). Role-based access control models. *Computer*, 29(2), 38–47. <https://doi.org/10.1109/2.485845>