Customizable Dashboards for QDAcity

BACHELOR THESIS

Hoang Pham Minh Khai

Submitted on 11 September 2023

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

Supervisor:
Julia Mucha, M.Sc
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

License

Abstract

QDAcity is a cloud-based web application that allows users to collaborate on qualitative data analysis. Qualitative data can come in an enormous amount of information and will need to be organized and managed. Therefore, qualitative data analysis software should provide proper tools for users to organize their data and optimize their workflow. However, the current implementation of the QDAcity dashboards is not competent enough in comparison to other qualitative data analysis software on the market. In the context of this thesis, we will analyze how we can design a dashboard that suits the users and present a solution for a new dashboard that can be customized accordingly for individual users. The implementation is required to be compatible with the current design of QDAcity, maintainable, and extendable.

Contents

List of Figures

List of Tables

List of Code Examples

Acronyms

FACR	First Acronym
QDA	Qualitative Data Analysist
CAQDAS	Computer assisted qualitative data analysis software
HTTPS	Hypertext Transfer Protocol Secure
REST	Representational state transfer
RESTful	A web API that obeys the REST constraints
GCE	Google Cloud Endpoint
GCP	Google Cloud Platform
FAQ	Frequently asked questions
PDF	Portable Document Format
RTF	Rich Text Format
RBAC	Role-based access control
API	Application programming interface
DAO	Data Access Object
GAE	Google App Engine
SPA	Single Page Application
CSR	Client Side Rendering
DOM	Document Object Model
PWA	Progressive Web Application
URL	Uniform Resource Locator
ID	Identifier

DB	Database

1 Introduction

1.1 Motivation

QDA software has evolved the field of qualitative research by providing powerful tools for organizing, analyzing, and interpreting a big amount of data. As researchers increasingly rely on QDA software to extract meaningful insights from textual and multimedia data, the design and usability of the software interfaces become paramount in ensuring efficient and effective data analysis.

An essential part of developing QDA Software is a dashboard design. Dashboard design plays an important role in user experience in QDA software, as it not only helps the users manage the projects they are working with, but also supports collaboration between researchers on the same projects, keeping track of the deadline, task tracking, and ensuring their progress. In particular, well-designed dashboards within QDA software can significantly enhance the user experience by offering intuitive navigation, visual representations of data, smooth interaction with analytical tools, and hence, increasing overall productivity.

The current QDAcity dashboard provides users with basic functionalities. However, we can improve the user experience by allowing them to be able to customize the dashboard matching with their needs. So each individual user, they can interpret their needed information faster. This thesis contributes to the dashboard, by making the dashboard easily configurable and intuitive to use. This thesis aims to investigate the principles and practices of dashboard design in QDA software, as well as they affect how the data analysis procedure. While analyzing the difference between the old dashboard and the new one and understanding the specific requirements, procedures, and cognitive processes involved in QDA, we will get an idea for creating a dashboard that has efficient navigation, effective visualization, and meaningful interpretation of data.

To address these challenges, this thesis will delve into the theoretical foundations, studies, and practical considerations of dashboard design within QDA software. By studying existing QDA software tools, and analyzing user requirements and preferences. Together with the student at the University of applied science Oberösterreich, we identify the key factors contributing to successful dashboard design in QDA.

1.2 Objectives

In the context of QDAcity, this thesis should explore dashboard design's theoretical principles and enable a modern-looking dashboard for researchers to complete their work. The process requires identifying the essential components, functionalities, and visualization that can enhance the usability and effectiveness of QDA software dashboard. The frontend is going to be customizable and together with the backend the configuration of the users can be stored in a database. The implementation should have some tests to control the functionality, as well as make sure it works if there is any update in components in the future. Also, the dashboard template should be generic so that it can be reused for the projects dashboard and the courses dashboard in the future.

1.3 Thesis structures

In Chapter 1 we describe the motive of the thesis, its objects, and its structure. Here we present the context of the research topic, our intent on what to achieve, and an overview of the thesis. Chapter 2 is about the most relevant related work and research for the thesis, we conduct a literature review, exploring studies and projects that are related to our thesis. Chapter 3 talks about the requirements for the thesis, here we will gather and specify the functional and non-functional requirements for our dashboard. Chapter 4 presents the architecture of our dashboard and its relevant parts, which are our detailed overview of the system structure, explaining the reasons for our choices, the different components, and their interconnections. In Chapter 5 we focus on the design and implementation of our reusable dashboard, we provide translation between requirements and architecture into a functioning dashboard, code examples, and diagrams. Chapter 6 gives us an evaluation of our system, which include self reflect on the requirements, as well as user feedback, usability tests will tell us how well the dashboard meets our user needs and identify areas for improvement. Finally, chapter 7 summarizes and concludes the thesis, highlights key findings from each chapter, and warp up important information.

2 Related work

In this chapter, we present a summary of related work that is relevant to this thesis, as well as identify tasks and appropriate functionalities for each dashboard. Initially, section 2.1 offers a basic overview of QDAcity. In section 2.2, we will analyze and discuss different types of dashboards and their impacts on the users, along with how we should implement them.

2.1 QDAcity

QDAcity is a single-page application (SPA) with client-side rendering (CSR) computer-assisted qualitative Data analysis software (CAQDAS) based in the cloud, providing tools and an environment for multiple analysts to work and collaborate. QDAcity is also a Progressive Web App (PWA)[1] since it's implementation contains Service Worker APIs[2] (according to Lygenda, 2022).

2.1.1 Component architecture of QDAcity

The frontend is implemented using Javascript with React library[3]. The backend is a RESTful web service built with Google's cloud infrastructure, called Google Cloud Platform (GCP)[4], including Google Cloud Endpoints frameworks[5] in Java, firestore database[6], cloud run[7]. These components communicate with each other via HTTPS requests.

[1]https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/What_is_a_progressive_web_app#progressive_web_apps

[2]https://web.dev/learn/pwa/service-workers/

[3]https://react.dev/

[4]https://cloud.google.com/

[5]https://cloud.google.com/endpoints

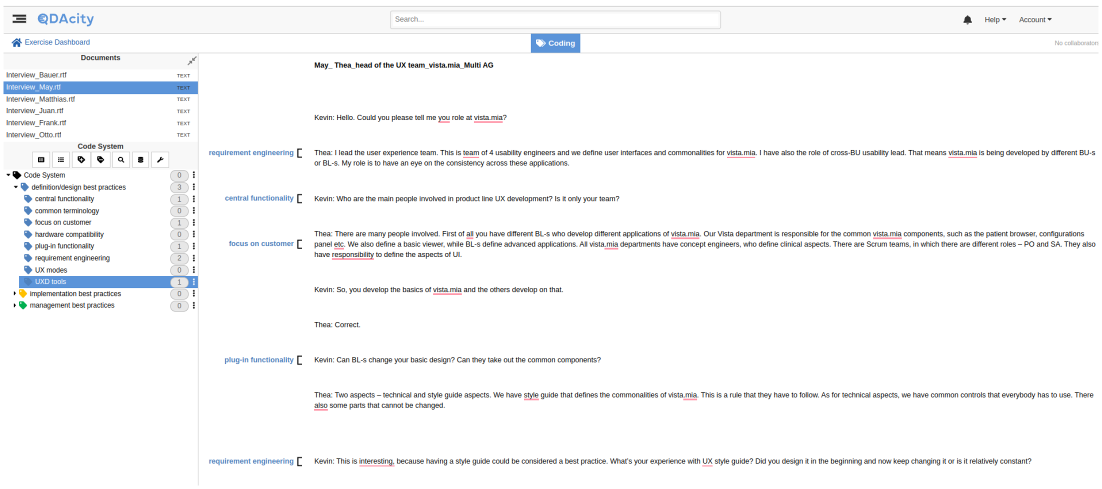
[6]https://cloud.google.com/firestore

[7]https://cloud.google.com/run

2.1.2 Features

QDAcity's wide range of features includes:

- Coding editor is the core of the tool, where a user can annotate segments of text (codings) with more abstract concepts (so-called code) with a document overview inside the code system (an organized structure of codes used to categorize and analyze qualitative data). Figure 2.1 shows a coding example of an exercise.



May_ Thea_head of the UX team_vista.mia_Multi AG

Kevin: Hello. Could you please tell me you role at vista.mia?

requirement engineering — Thea: I lead the user experience team. This is team of 4 usability engineers and we define user interfaces and commonalities for vista.mia. I have also the role of cross-BU usability lead. That means vista.mia is being developed by different BU-s or BL-s. My role is to have an eye on the consistency across these applications.

central functionality — Kevin: Who are the main people involved in product line UX development? Is it only your team?

focus on customer — Thea: There are many people involved. First of all you have different BL-s who develop different applications of vista.mia. Our Vista department is responsible for the common vista.mia components, such as the patient browser, configurations panel etc. We also define a basic viewer, while BL-s define advanced applications. All vista.mia departments have concept engineers, who define clinical aspects. There are Scrum teams, in which there are different roles – PO and SA. They also have responsibility to define the aspects of UI.

Kevin: So, you develop the basics of vista.mia and the others develop on that.

Thea: Correct.

plug-in functionality — Kevin: Can BL-s change your basic design? Can they take out the common components?

Thea: Two aspects – technical and style guide aspects. We have style guide that defines the commonalities of vista.mia. This is a rule that they have to follow. As for technical aspects, we have common controls that everybody has to use. There also some parts that cannot be changed.

requirement engineering — Kevin: This is interesting, because having a style guide could be considered a best practice. What's your experience with UX style guide? Did you design it in the beginning and now keep changing it or is it relatively constant?

Figure 2.1: QDAcity coding editor of an exercise

- Import/export data in various formats like PDF-, RTF-, and interview media data (figure 2.2).

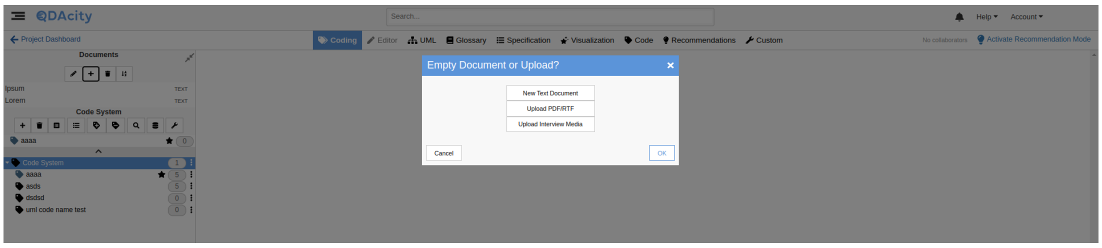
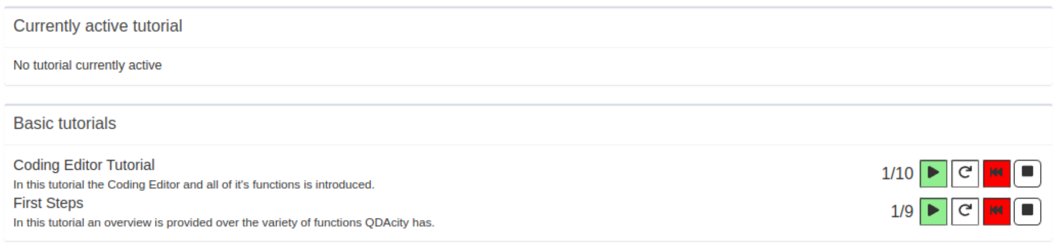


Figure 2.2: QDAcity create/upload document window

- Tutorials for beginners and FAQ sections (figure 2.3).



Currently active tutorial

No tutorial currently active

Basic tutorials

Coding Editor Tutorial
In this tutorial the Coding Editor and all of it's functions is introduced. 1/10

First Steps
In this tutorial an overview is provided over the variety of functions QDAcity has. 1/9

Figure 2.3: QDAcity tutorial

- Project revision history (snapshots of the state of a project) (figure 2.4).

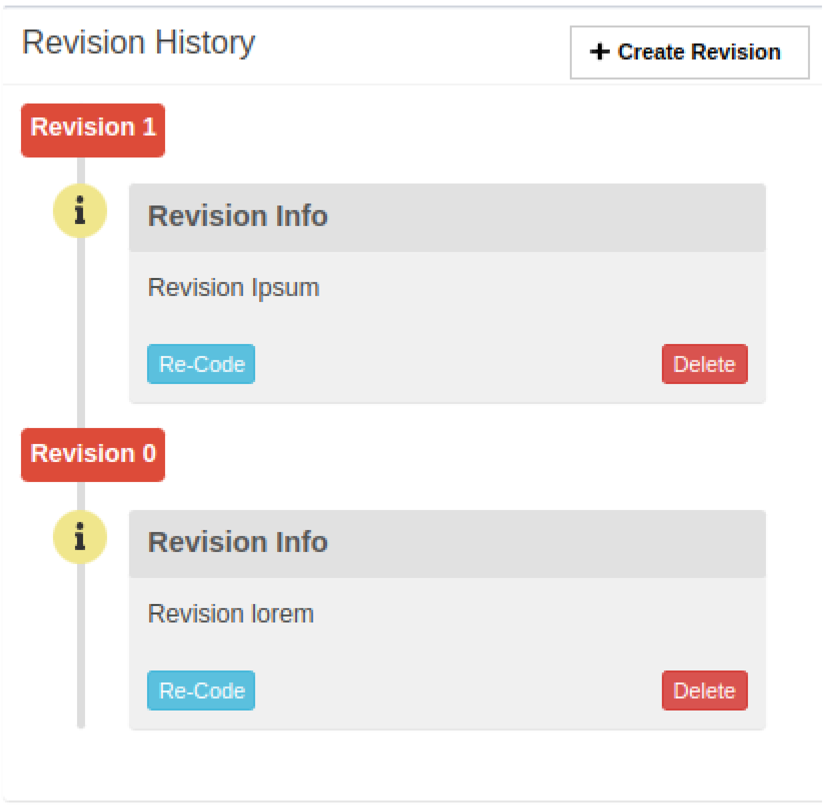
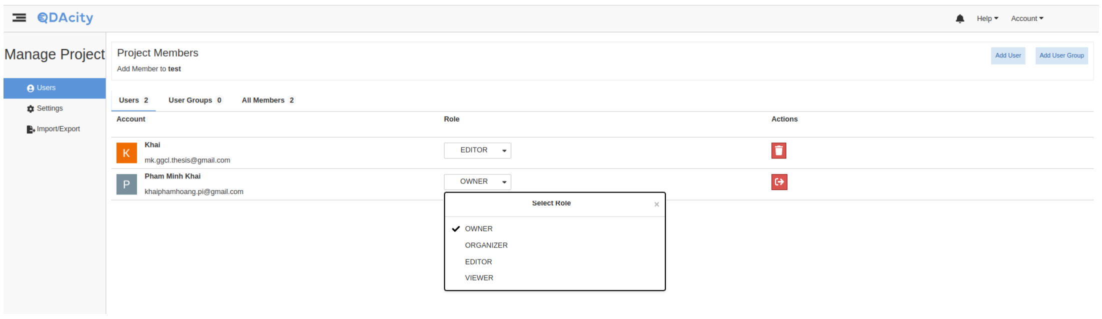


Figure 2.4: QDAcity revision history

- Role-based access control (RBAC), permissions are assigned to users by roles, allowing multiple users to work on shared projects (figure 2.5).



Project Members
Add Member to **test**

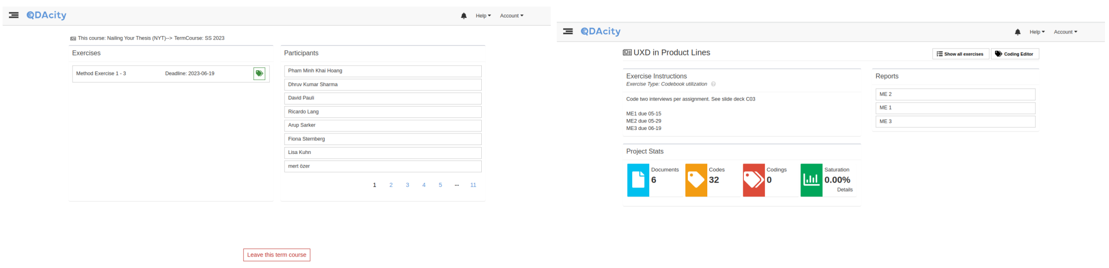
Users 2 | User Groups 0 | All Members 2

Account	Role	Actions
Khai mk.ggcl.thesis@gmail.com	EDITOR	
Pham Minh Khai khaiphamhoang.pi@gmail.com	OWNER	

✓ OWNER
ORGANIZER
EDITOR
VIEWER

Figure 2.5: QDAcity role-based access control

- Course-system to support the teaching of QDA. (figure 2.6)



This course: Nailing Your Thesis (NYT)--> TermCourse: SS 2023

Exercises
Method Exercise 1 - 3 Deadline: 2023-06-19

Participants
Pham Minh Khai Hoang
Dhruv Kumar Sharma
David Pauli
Ricardo Lang
Arup Sarker
Fiona Sternberg
Lisa Kuhn
mert özer

1 2 3 4 5 ... 11

(a) QDAcity exercises list view

UXD in Product Lines

Exercise Instructions
Exercise Type: Codebook utilization

Code two interviews per assignment. See slide deck C03

ME1 due 05-15
ME2 due 05-29
ME3 due 06-19

Reports
ME 2
ME 1
ME 3

Project Stats
Documents 6
Codes 32
Codings 0
Saturation 0.00%

(b) QDAcity exercise dashboard

Figure 2.6: QDAcity course-system

- Dashboards that display information, the information depends on the type of dashboard, we have three different dashboards: Personal, project, and course. These are the main focus of the thesis and will be explained in more detail in section 2.3.

Our focus of this thesis is to make the dashboard customizable and design a new visual for it. To do that, we will first do research on other QDA software in section 2.2.

2.2 Analyse QDA software dashboards

To design the QDAcity dashboard, we are going to do research on other QDA software on the market[8], a look at how the QDA software dashboard looks like, and how they are structured.

There is a design rule created in 2001 by Jeffrey Zeldman[9], called Three-click-rule, states that "a user should always be able to find their desired information within

[8]https://renaissancerachel.com/best-qualitative-data-analysis-software/
[9]https://tillerdigital.com/glossary/3-click-rule/

three clicks or less"[10]. However, another study shows that "if the progressive revelation of information takes the user down a path towards refinement that feels like progress and gets them where they need to be, they will give you up to twelve clicks before turning grumpy" (Shah, 2014). So it's important for us to also analyze the task flows, from the moment users log in until they start coding each dashboard.

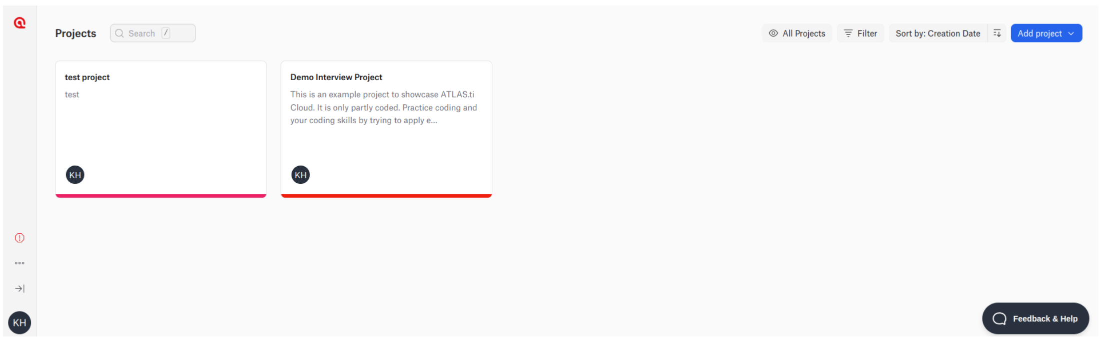
As a result, we will draw some insights taken from the analysis of these dashboards.

2.2.1 Investigate other QDA software dashboards

ATLAS.ti[11]

When first logging in to Atlas.ti, the user will have access to a projects dashboard, each project is presented as a card (see Figure 2.7).

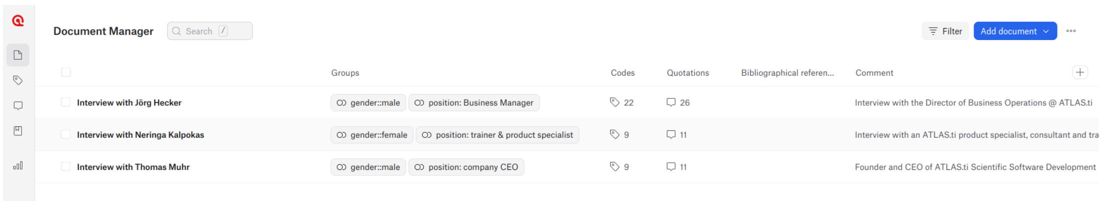
These cards can be opened and will redirect the user to the project dashboard. The user can now access the document manager with an overview of the project (figure 2.8). Additionally, there is a codes manager that presents all the codes (code in some other QDA software is also called tag) of that project (figure 2.9). The user can now open a document and start coding/tagging the information.



test project
test

Demo Interview Project
This is an example project to showcase ATLAS.ti Cloud. It is only partly coded. Practice coding and your coding skills by trying to apply e...

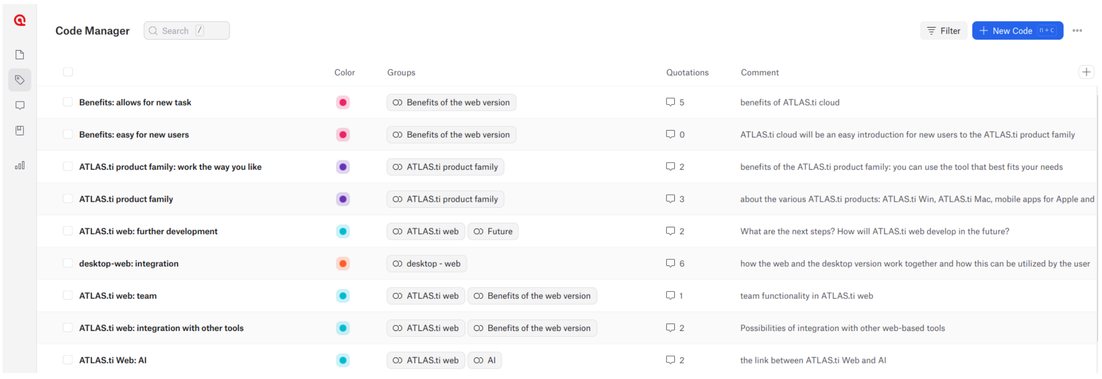
Figure 2.7: Atlas.ti projects dashboard



	Groups	Codes	Quotations	Bibliographical referen...	Comment
Interview with Jörg Hecker	gender::male position: Business Manager	22	26		Interview with the Director of Business Operations @ ATLAS.ti
Interview with Neringa Kalpokas	gender::female position: trainer & product specialist	9	11		Interview with an ATLAS.ti product specialist, consultant and tra
Interview with Thomas Muhr	gender::male position: company CEO	9	11		Founder and CEO of ATLAS.ti Scientific Software Development

Figure 2.8: Atlas.ti document manager

[10]https://brand-experience.ieee.org/the-3-click-rule-myth-or-fact/
[11]https://atlasti.com/



	Color	Groups	Quotations	Comment
Benefits: allows for new task	●	Benefits of the web version	5	benefits of ATLAS.ti cloud
Benefits: easy for new users	●	Benefits of the web version	0	ATLAS.ti cloud will be an easy introduction for new users to the ATLAS.ti product family
ATLAS.ti product family: work the way you like	●	ATLAS.ti product family	2	benefits of the ATLAS.ti product family: you can use the tool that best fits your needs
ATLAS.ti product family	●	ATLAS.ti product family	3	about the various ATLAS.ti products: ATLAS.ti Win, ATLAS.ti Mac, mobile apps for Apple and
ATLAS.ti web: further development	●	ATLAS.ti web, Future	2	What are the next steps? How will ATLAS.ti web develop in the future?
desktop-web: integration	●	desktop - web	6	how the web and the desktop version work together and how this can be utilized by the user
ATLAS.ti web: team	●	ATLAS.ti web, Benefits of the web version	1	team functionality in ATLAS.ti web
ATLAS.ti web: integration with other tools	●	ATLAS.ti web, Benefits of the web version	2	Possibilities of integration with other web-based tools
ATLAS.ti Web: AI	●	ATLAS.ti web, AI	2	the link between ATLAS.ti Web and AI

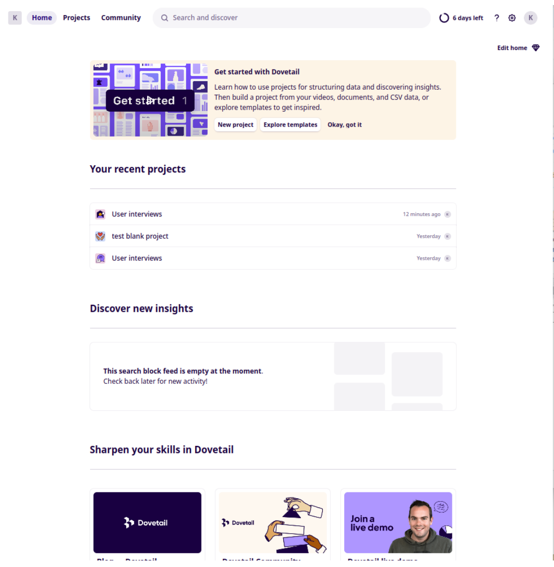
Figure 2.9: Atlas.ti codes manager

Atlas.ti has a straightforward task flow, directly navigating user to the important section of their software, which is the coding of the documentation. Additionally, a lot of information is provided, but they still leave a lot of space and don't overwhelm users with too much information.

Dovetail[12]

In Dovetail each user is provided with two dashboards (figure 2.10), a home dashboard, and a projects dashboard. A home dashboard by default presents the recently opened projects by the user, a search block feed for new insights into the software, and a suggestion tutorial/guide. These sections are placed below each other and the user can edit them when upgrading to premium. For the projects dashboard, there are two views, one for the organized folders and one for all the projects of a user.

[12]https://dovetail.com/



Get started with Dovetail

Learn how to use projects for structuring data and discovering insights. Then build a project from your videos, documents, and CSV data, or explore templates to get inspired.

Your recent projects

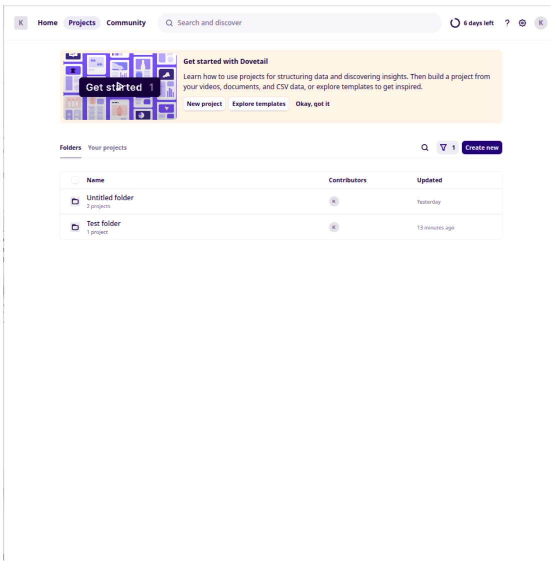
User interviews	12 minutes ago
test blank project	Yesterday
User interviews	Yesterday

Discover new insights

This search block feed is empty at the moment.
Check back later for new activity!

Sharpen your skills in Dovetail

(a) Dovetail personal dashboard



Get started with Dovetail

Learn how to use projects for structuring data and discovering insights. Then build a project from your videos, documents, and CSV data, or explore templates to get inspired.

Name	Contributors	Updated
Untitled folder 2 projects	K	Yesterday
Test folder 1 project	K	13 minutes ago

(b) Dovetail projects dashboard (list view)

Figure 2.10: Dovetail dashboards

Similar to Atlas.ti, Dovetail also offers a tag dashboard to present all the tags of the projects, these tags can be organized into groups, with each group card can be visually distinguished by colors (figure 2.11).

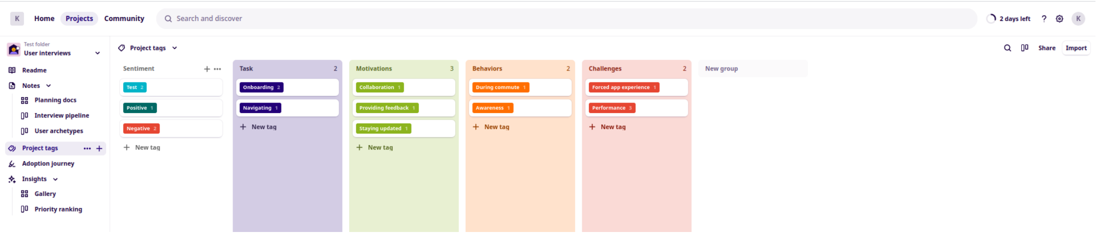


Figure 2.11: Dovetail tags dashboard

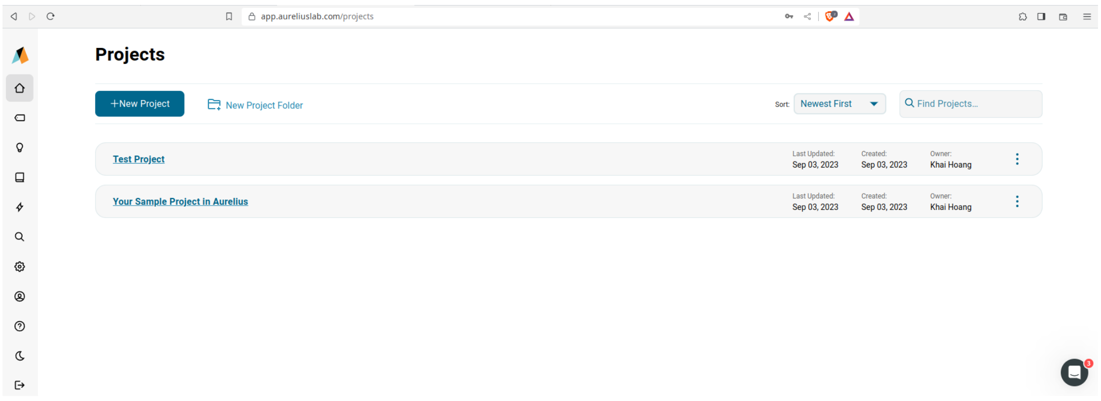
Once users log in, they can access the project directly with the recently opened project functionality, or create a new one in the projects list view. Once a project is selected the user is directed to the notes (similar to documents in QDAcity software), select a note then the user can start coding/tagging. Dovetail also has a simple and direct workflow, making an easy for the user to use their software.

Aurelius[13]

Aurelius has a similar project dashboard design to Dovetail with a project list (figure 2.12), some additional information about the project like the last updated

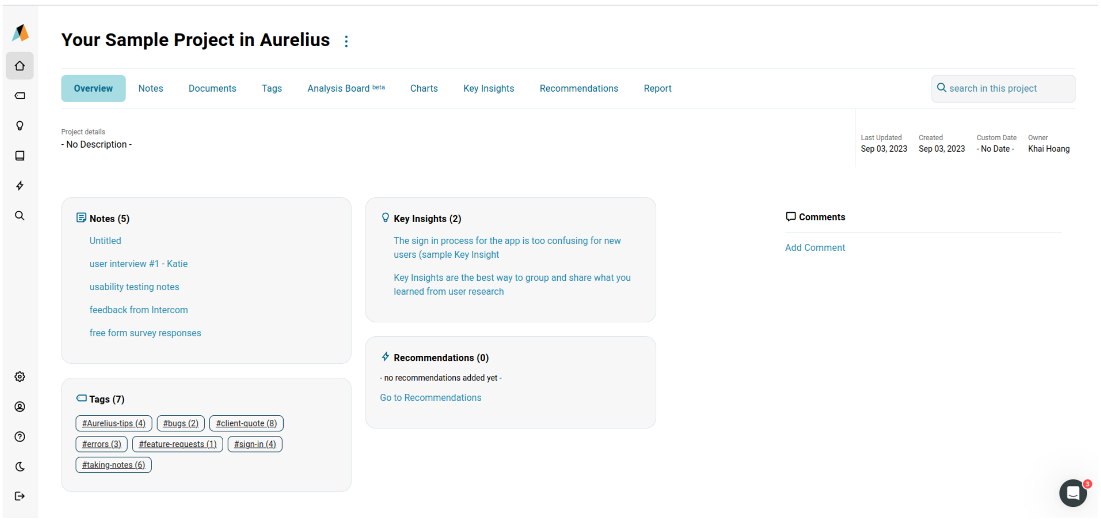
[13] https://www.aureliuslab.com/

time, and a search bar. From here user can navigate to the dashboard of each project to have an overview of the project (figure 2.13). Aurelius also has a tags dashboard to keep track of all tags. To start coding, the user needs to navigate to the notes area, and then open/create a note. In comparison, Aurelius has a basic task flow, but it can create user confusion due to its use of similar terminology for both notes and documents.



	Last Updated:	Created:	Owner:
Test Project	Sep 03, 2023	Sep 03, 2023	Khai Hoang
Your Sample Project in Aurelius	Sep 03, 2023	Sep 03, 2023	Khai Hoang

Figure 2.12: Aurelius projects list



Project details
- No Description -

Last Updated	Created	Custom Date	Owner
Sep 03, 2023	Sep 03, 2023	- No Date -	Khai Hoang

Notes (5)
Untitled
user interview #1 - Katie
usability testing notes
feedback from Intercom
free form survey responses

Key Insights (2)
The sign in process for the app is too confusing for new users (sample Key Insight
Key Insights are the best way to group and share what you learned from user research

Recommendations (0)
- no recommendations added yet -

Tags (7)
#Aurelius-tips (4) #bugs (2) #client-quote (8) #errors (3) #feature-requests (1) #sign-in (4) #taking-notes (6)

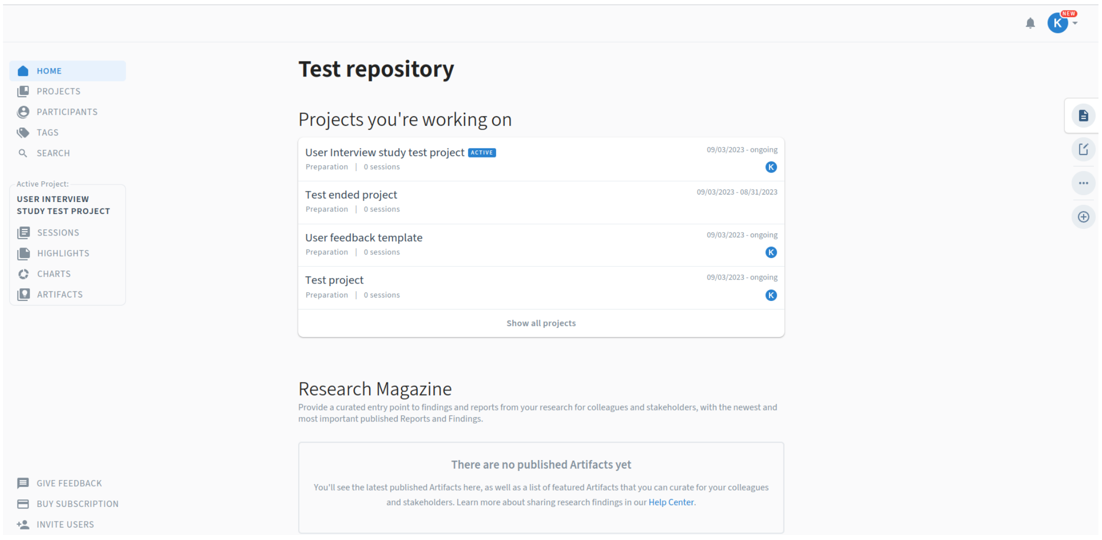
Figure 2.13: Aurelius project dashboard

Condens[14]

Another QDA software dashboard here to analyze has a lot in common with the others. Condens offers the user the following dashboards, a home dashboard, a project dashboard, a participants dashboard, and a tags dashboard. They all have

[14]https://app.condens.io/

a list view for their items (figure 2.14). The tags dashboard has an additional function that allows user to group their tag into cards for easier management (figure 2.15). Using drag and drop functionality, the user can easily change the position of the cards to fit their priority, moving the tags between cards. The right sidebar can be expanded or collapsed based on the user's need and provides additional tools for users, such as pinning the dashboard, adding notes, and navigating (figure 2.16).



Test repository

Projects you're working on

User Interview study test project ACTIVE — Preparation | 0 sessions — 09/03/2023 - ongoing

Test ended project — Preparation | 0 sessions — 09/03/2023 - 08/31/2023

User feedback template — Preparation | 0 sessions — 09/03/2023 - ongoing

Test project — Preparation | 0 sessions — 09/03/2023 - ongoing

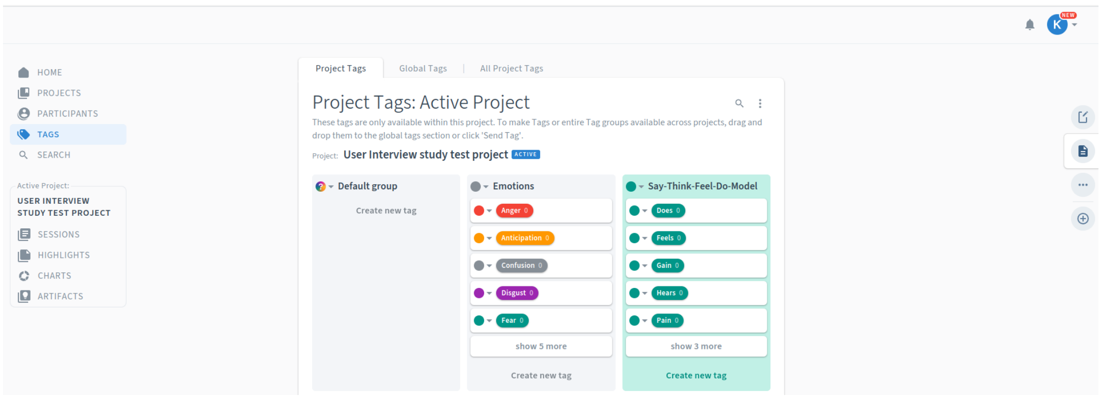
Research Magazine

Provide a curated entry point to findings and reports from your research for colleagues and stakeholders, with the newest and most important published Reports and Findings.

There are no published Artifacts yet

You'll see the latest published Artifacts here, as well as a list of featured Artifacts that you can curate for your colleagues and stakeholders. Learn more about sharing research findings in our Help Center.

Figure 2.14: Condens home dashboard

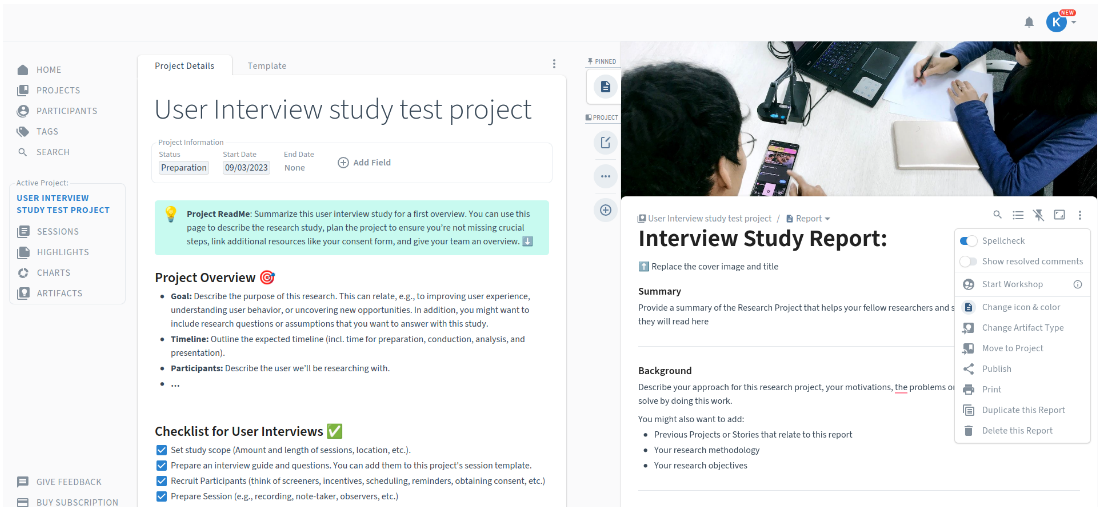


Project Tags: Active Project

These tags are only available within this project. To make Tags or entire Tag groups available across projects, drag and drop them to the global tags section or click 'Send Tag'.

Project: User Interview study test project ACTIVE

Figure 2.15: Condens tags dashboard



User Interview study test project

Project Information

Status	Start Date	End Date	
Preparation	09/03/2023	None	Add Field

💡 **Project ReadMe**: Summarize this user interview study for a first overview. You can use this page to describe the research study, plan the project to ensure you're not missing crucial steps, link additional resources like your consent form, and give your team an overview. ⬇️

Project Overview 🎯

- **Goal:** Describe the purpose of this research. This can relate, e.g., to improving user experience, understanding user behavior, or uncovering new opportunities. In addition, you might want to include research questions or assumptions that you want to answer with this study.
- **Timeline:** Outline the expected timeline (incl. time for preparation, conduction, analysis, and presentation).
- **Participants:** Describe the user we'll be researching with.
- ...

Checklist for User Interviews ✅

- ✓ Set study scope (Amount and length of sessions, location, etc.).
- ✓ Prepare an interview guide and questions. You can add them to this project's session template.
- ✓ Recruit Participants (think of screeners, incentives, scheduling, reminders, obtaining consent, etc.)
- ✓ Prepare Session (e.g., recording, note-taker, observers, etc.)

User Interview study test project / Report

Interview Study Report:

⬆️ Replace the cover image and title

Summary

Provide a summary of the Research Project that helps your fellow researchers and s they will read here

Background

Describe your approach for this research project, your motivations, the problems o solve by doing this work.

You might also want to add:

- Previous Projects or Stories that relate to this report
- Your research methodology
- Your research objectives

Figure 2.16: Condens project with right sidebar expanded

Condesn requires a bit more navigation to start coding. The user has to choose a project, instead of being navigated to the documents/notes list view as other QDA software, a README about the project is opened. The user will have to click on "SESSIONS" in the left sidebar to open/create a document/note to start coding. This has the advantage of giving more information to users, but it doesn't quite follow the prototype of other QDA software, making it a bit hard approach for new users.

2.2.2 Conclusion from the research

We can find some common in these dashboards that we analyze. From a functionality perspective, they all have a list of projects view to present all the projects, along with a filtering mechanism. In addition, these software applications separate the document/note manager with code manager into two views for managing them. In QDAcity, when the project expands over time, adding more documents and codes can potentially overwhelm the user with an increasing amount of information displayed on the screen. Also, the documents and codes also take up more space in the coding editor, reducing the visual clarity of coding tasks. Therefore, we will design a separate document management and code management view in the future. From the UI we can see that they have centralized their tools, and the box/card's corners are rounded. Among the four, three QDA software are using a left sidebar for navigating. Furthermore, we can see that they have a lot of space between them, group up their information, and create a highly prototypical site. That is also something we can notice when designing a new look for QDAcity.

2.3 QDAcity Dashboard

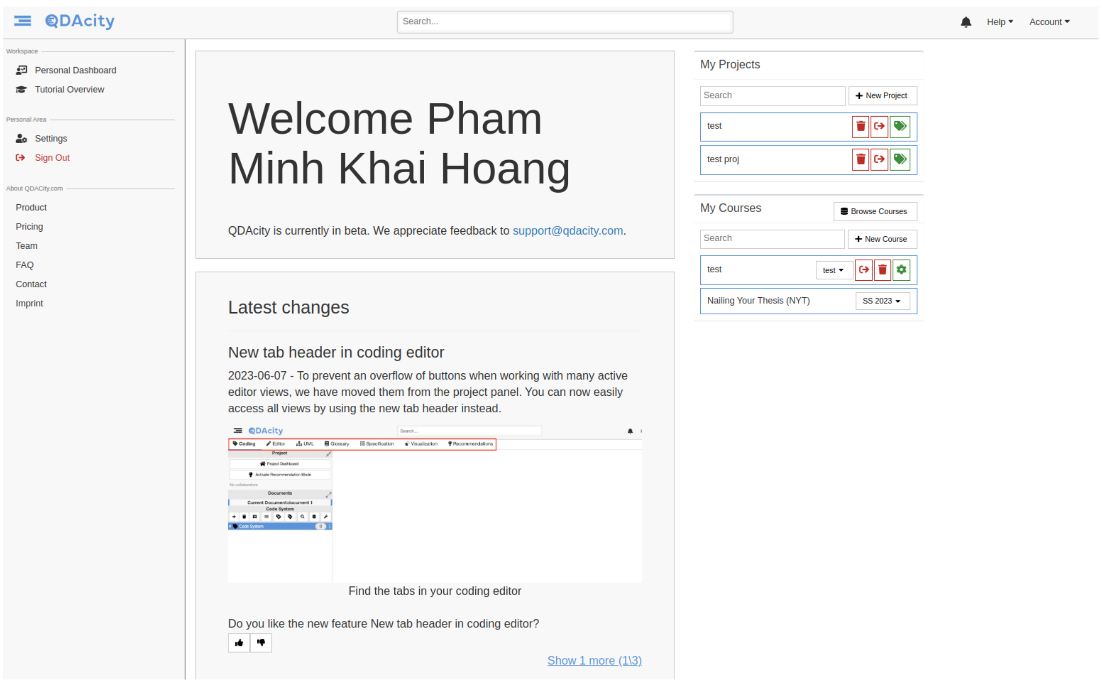
In this section, we are going to present the implementation of a dashboard, how a modern dashboard should look in general, and how we can give it a better look. Additionally, we are going to analyze how a dashboard interacts with the users, in our case they are researchers, students, and teachers.

2.3.1 Personal Dashboard

This dashboard should be a default for newly registered users, where she or he can see and manage projects, access courses, and get news about the application.

The current personal dashboard

In the current state, a screenshot of the current online version of QDAcity with test projects and courses was taken as a reference (figure 2.17).



Welcome Pham Minh Khai Hoang

QDAcity is currently in beta. We appreciate feedback to support@qdacity.com.

Latest changes

New tab header in coding editor

2023-06-07 - To prevent an overflow of buttons when working with many active editor views, we have moved them from the project panel. You can now easily access all views by using the new tab header instead.

Find the tabs in your coding editor

Do you like the new feature New tab header in coding editor?

Show 1 more (1\3)

Figure 2.17: Old Personal Dashboard

As we can see the personal dashboard has a very basic approach. The logical layout does not follow the Inverted Pyramid design principle (Sisense, 2023). The concept of this is to divide the contents into three tiers, ranked in decreasing order of importance. The most significant information should be placed at the top, followed by details that help provide more context, and at the bottom the information there is general and background information that offers more detail and allows users to dive deeper(as seen in figure 2.18). Our old dashboard has but a huge welcome and latest changes panel in the middle of the dashboard. This might be interesting for first-time sign-in users, but ultimately they are not the most noteworthy information for the users. Since their most important concern here is the projects and courses area, we should have this area placed at the top of the dashboard. In the old one they are but a very small part on the right side of the dashboard.

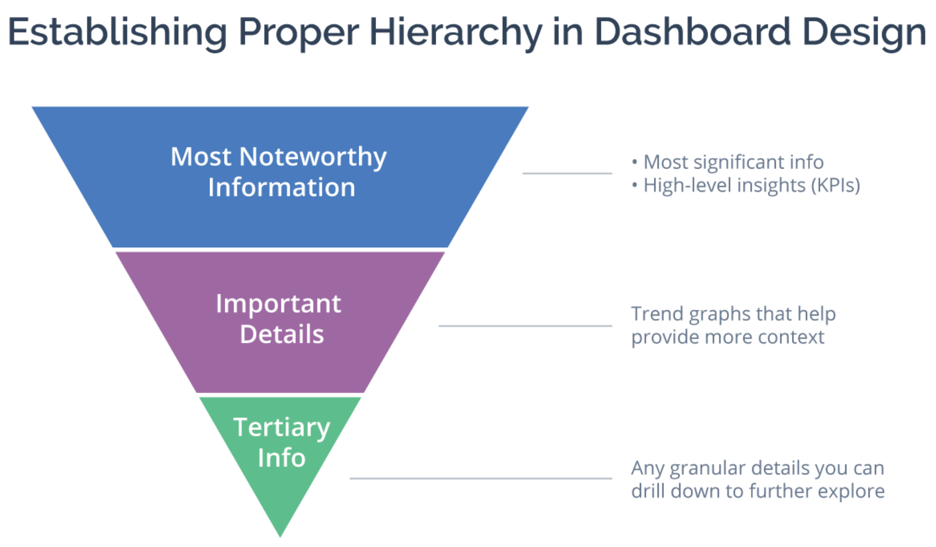

Establishing Proper Hierarchy in Dashboard Design
Most Noteworthy Information
Important Details
Tertiary Info
• Most significant info
• High-level insights (KPIs)
Trend graphs that help provide more context
Any granular details you can drill down to further explore

Figure 2.18: Logic Layout: The Inverted Pyramid [14]

The new personal dashboard

Here we have screenshots for the first approach of designing the dashboard with the cards and how the dashboards will look with the result working of the design of the student project. Figure 2.19 illustrates the idea of cards in two variations, one with a sharp corner and the other with a rounded corner.

[14]https://datameaning.com/dashboard-design/dashboard-design-best-practices/

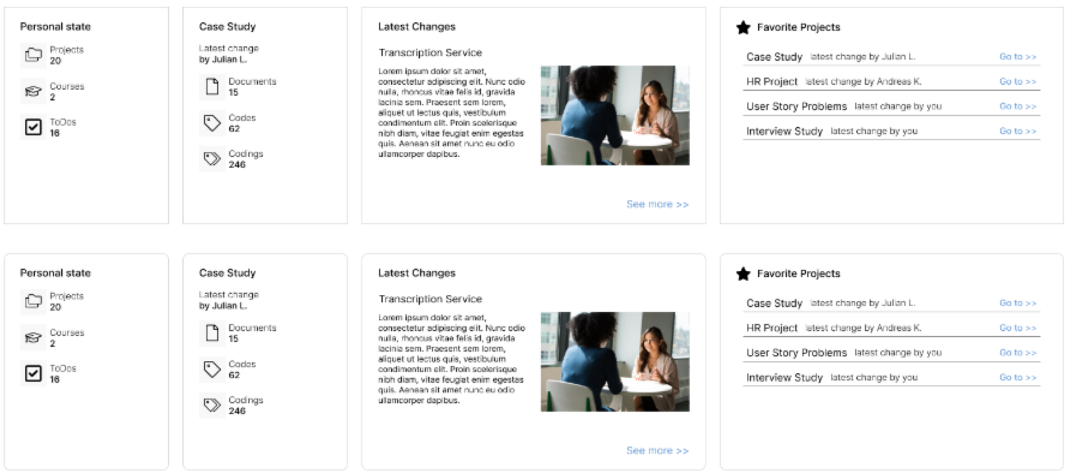

Personal state
Projects 20
Courses 2
ToDos 16
Case Study
Latest change by Julian L.
Documents 15
Codes 62
Codings 246
Latest Changes
Transcription Service
See more >>
Favorite Projects
Case Study latest change by Julian L. Go to >>
HR Project latest change by Andreas K. Go to >>
User Story Problems latest change by you Go to >>
Interview Study latest change by you Go to >>

Figure 2.19: Two versions of cards

Our initial idea for designing the dashboard is to use cards for different types of content, they have a cleaner look and the users can configure their own cards easily. We also want to improve this by following the visual hierarchy, "[...] the most important content should stand out the most, and the least important should stand out the least", and the visual flow, "[...] a well-designed visual hierarchy sets up focal points on the page wherever you need to draw attention to the most important elements, and visual flow leads the eyes from those points into the less-important information" (Tidwell, 2005). Therefore, we will create some initial most important cards with bigger card sizes for our users, for example, the projects and courses cards. We also value a minimalist design, which is not only a trend for website design(Abbas, 2023), but also plays a role in our cognitive fluency and visual information processing (Walker, 2022). According to Miller, 1956, the average adult brain's short-term memory (where people temporarily store and process information) is limited between five to nine "chunks" of information (a chunk is the most significant and recognizable unit in the presented material that the person recognizes). Information overload can result in diverting attention, distraction, and decreased productivity (Malak, 2022). Additionally, a study from Google has also shown that users prefer websites with low visual complexity and high prototypicality[15]. Therefore, we want our design to be simple, less visually complex, follow the QDA software layout theme, and be open using spaces. Another aspect affecting user experience is the loading speed (Georgiou, 2014). According to the findings of Anderson (Anderson, 2023), 47 percent of consumers expect a page to load in two seconds or less, this also shows that a fast-loading site is a crucial factor in web design.

[15]https://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/38315.pdf

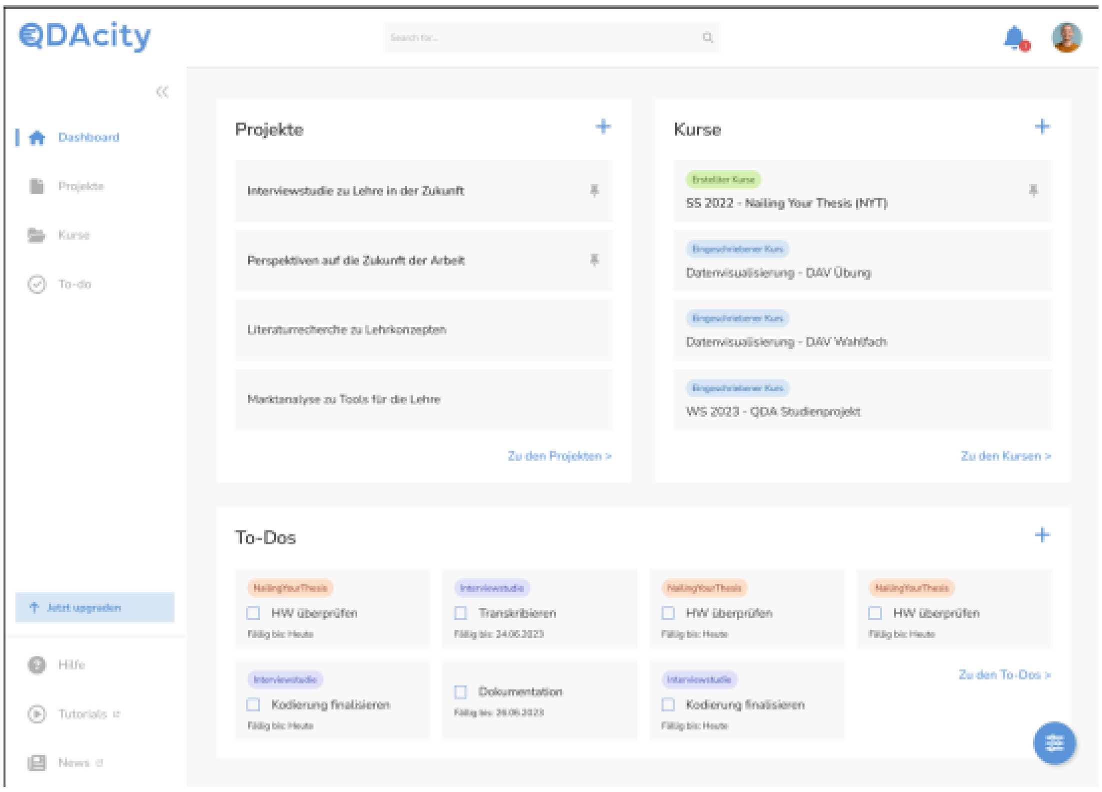


Figure 2.20: Dashboard design from the student from FHOÖ

This new design (figure 2.20) not only follows the design principles but also provides a modern-looking dashboard with quick navigation and interpretation of information. Our design and implementation will be based on this design and the analysis of other tools.

2.3.2 List of projects view

A list of projects view can be accessed by selecting an all-project link on the personal dashboard (the navigation structure can be seen in subsection 2.3.4). Here a list of all projects in which a user participates, irrespective of their designated role within the project, is presented. In addition, a list of filters is provided so that the user can effectively retrieve their preferred projects. Project details can be accessed by either clicking on the project in the Personal Dashboard or selecting a project in the list of projects view. The users will be redirected to the project dashboard, which provides an overview of the project and contains management functions and metrics about the project. It is intended to serve as a collaboration point for several project participants to exchange information or inform each other.

2.3.3 Project Dashboard

QDAcity has a project dashboard that displays information static information about the project and manages the user, to-do, and revision (see figure 2.21). From here the user can access the coding editor, where users can start coding (2.22). Here the coding editor also displays the information about the documents of that project and all code (in other QDA software also called tag) created in the code system and their frequency of utilization.

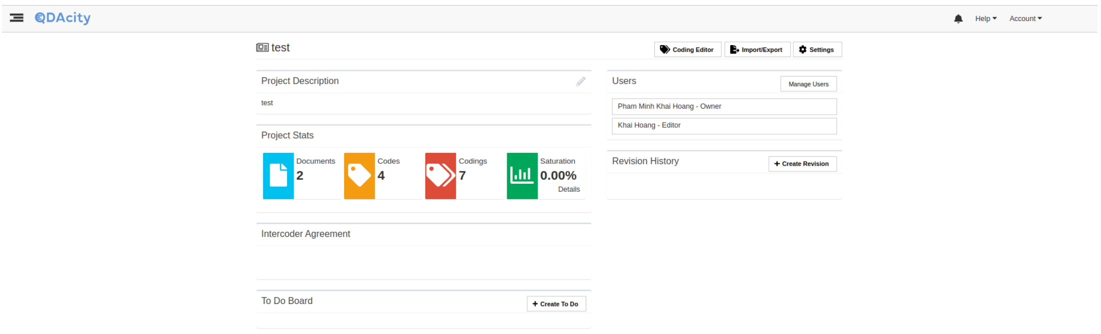
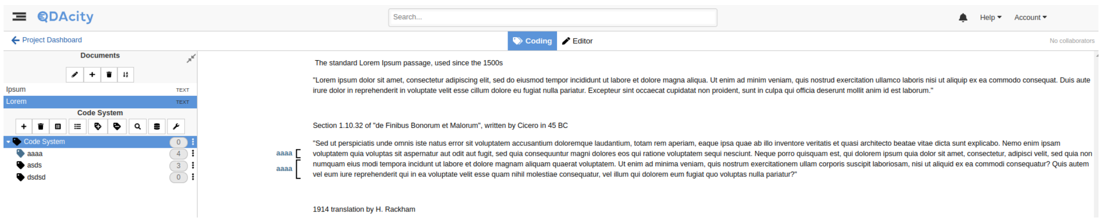


Figure 2.21: Qdacity Project Dashboard



The standard Lorem Ipsum passage, used since the 1500s

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Section 1.10.32 of "de Finibus Bonorum et Malorum", written by Cicero in 45 BC

"Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?"

1914 translation by H. Rackham

Figure 2.22: Qdacity Coding Editor

2.3.4 Navigation structure

To be able to use the personal dashboard, the user must first register an account in QDAcity. This can be done via Email/Password Registration or with Google OAuth 2.0[16]. After registration, the users will be directed to the personal dashboard and have access to their personal dashboard. From there the user will have access to different pages, such as the list of projects view, list of courses view, and news-feed site updating news about the development/functionalities of QDAcity. The cards should provide a link that leads to a more specific site about the information being displayed on the cards. From the list of projects/courses view, users can select a project/course to get more detailed information about them, get access to the coding editor, or get back to the personal dashboard.

[16]https://developers.google.com/identity/protocols/oauth2

The concept is to use different pieces of information layer to reduce the amount of information as well as the complexity of each site, therefore enhancing the readability of them. Figure 2.23 shows the navigation of the dashboard.

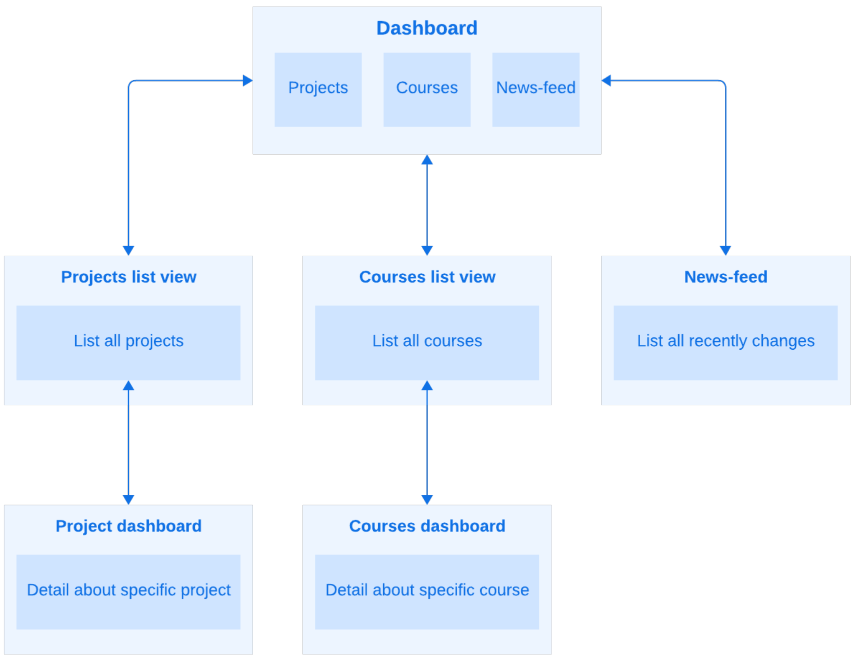


Figure 2.23: Navigation Design

3 Requirement

This chapter states the requirements for the new personal dashboard, divided into 2 categories, the functional requirements (FR) and nonfunctional requirements (NFR) using the template provided by Rupp and SOPHISTGesellschaft für Innovatives Software-Engineering (SOPHIST, 2016). In each section, we will have a brief description of the template before jumping into the details of the requirements.

3.1 Functional Requirement

The syntax and semantics of the FR follow the template FunctionalMASTeR as seen in the figure below.

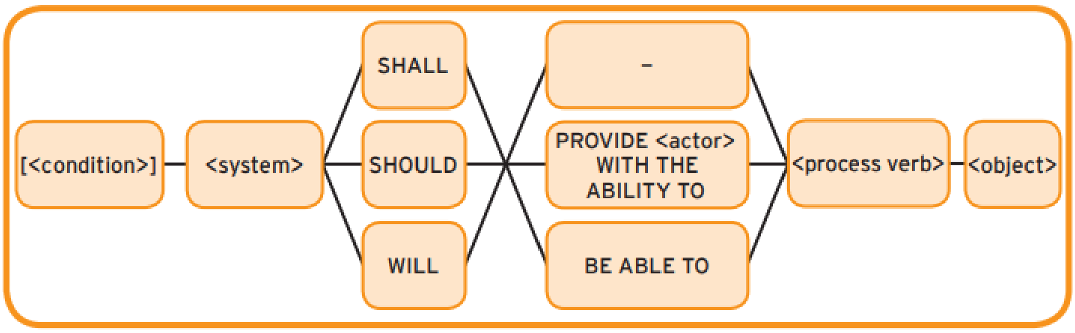


Figure 3.1: Functional Requirements Template

FR sentences will be structured based on this template as a blueprint with five steps total. Starting with our system, which is given, we define in step one the importance of the system's functionality with three keywords "shall", "should", and "will" in descending order before defining that functionality in step 2 in process verb. We will then specify the type of functionality in step 3 and define our object in step 4. The condition in step 5, by putting in a square bracket is defined as optional and can be added using the following syntax. The condition is put in a square bracket, which means it is optional for the requirements.

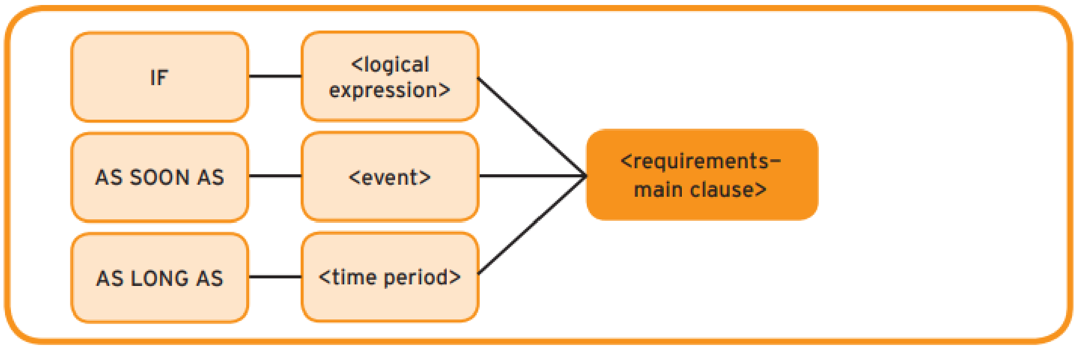


Figure 3.2: Condition of functional requirements

The conditions template is categorized into three groups. Conditions based on logical expressions using the keyword "If", conditions triggered by events using the keyword "As soon as", and conditions tied to a specific time using the keyword "As long as".

We will proceed to label the FR following the above-explained template.

FR-1 If the user logs in for the first time, the personal dashboard shall provide the user a personal dashboard with two default project and course cards.

FR-2 The personal dashboard shall provide users with the ability to save their dashboard configuration.

FR-3 As soon as users log in to QDA, the personal dashboard shall provide users with the ability to retrieve the dashboard configuration they saved before.

FR-4 If users enter edit mode, the dashboard shall provide users with the ability to add/remove/move cards.

FR-5 The projects list view shall be able to present all the projects.

FR-6 If the user wants to have a view of only pinned projects, the projects list view should be able to filter only pinned projects.

FR-7 If there are projects that the user frequently uses, these projects shall be able to be pinned to the pinned projects card in the personal dashboard.

FR-8 If users want to navigate between the personal dashboard, projects list view, and project dashboard, the dashboard system shall provide the user a clickable link to navigate between them.

FR-9 If users follow a course/project, they are able to turn on the notification in customized dashboard mode by clicking on the alarm icon, so that they will receive news about that course/project.

FR-10 If the project is pinned by a user, the database shall be able to store the pinned project information only for the user who pinned it.

FR-11 If the pinned project is deleted, the database shall delete the information about the pinned project.

FR-12 If the user profile is deleted, the dashboard settings profile shall be deleted.

3.2 Non-Functional Requirement

The operating environment of the dashboard is formulated as shown in the figure, 3.3, based on the template by MASTER – Schablonen für alle Fälle (SOPHIST, 2016)

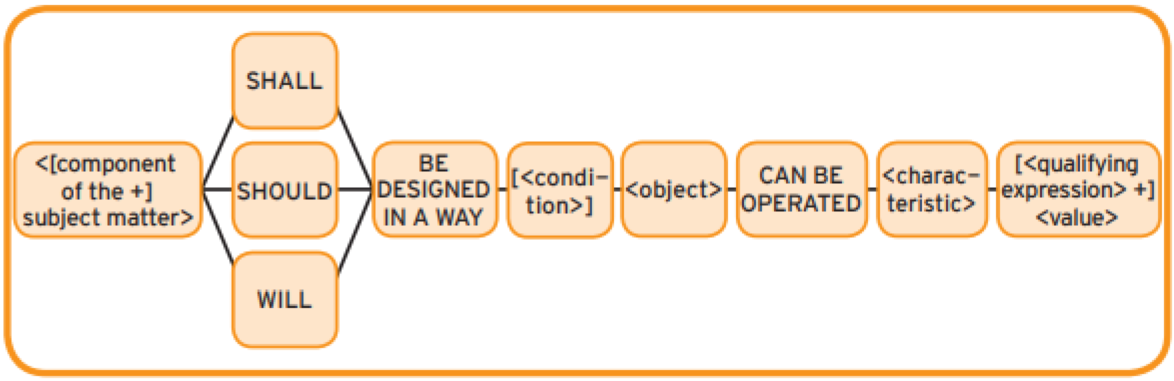


Figure 3.3: EnvironmentMASTeR

NFR-1 The dashboard should be designed in a way that can be used easily by the users without the need for any tutorials.

NFR-2 The dashboard should be designed in a way that allows users to quickly access projects/courses and interpret relevant information to the project/-courses such as their titles or user's role in that project/course.

NFR-3 The buttons in the dashboard should be designed in a way that interacts with users and helps people use their functionality by providing a tooltip.

NFR-4 The components and styling should be designed in a way that is compact for different web browsers, such as Google Chrome, Mozilla Firefox, and Brave.

NFR-5 The color scheme should be designed in a way that matches the color scheme of QDAcity.

NFR-6 The dashboard settings of each user shall be designed in a way that can only be modified by its owner, which means the cards can be added/deleted/positioned only by the user who created them.

NFR-7 The dashboard should be designed in a way that is easily reusable for project and course dashboards.

NFR-8 QDA Dashboard should be designed in a way that gives users a good user experience, including the intuitiveness of the dashboard, ease of navigation between projects, and the ability to quickly access relevant information.

NFR-9 The Dashboard must be designed in a way that will adjust responsively to the width of other devices outside of a computer such as a phone or tablet.

4 Architecture

The dashboard architecture follows the design of QDAcity software in general, the dashboard frontend is going to be implemented in Javascript language with React library. Our backend and database are implemented with services provided by GCP[1]. Here we are using GCE platform as an extensible service proxy to develop, deploy, and manage APIs in the backend. For the database, we are going to use Google Cloud Storage[2] and Redis[3].

4.1 Initial architecture of the dashboard

The dashboard consists of several key components, including a welcome panel, a list of projects and courses, a news feeds panel, and a navigation bar. However, the current distribution of space is not optimal, as it devotes too much area to the welcome and news feed panels. Considering that the primary purpose of the software is to access projects and courses and perform analysis tasks, this imbalance needs to be addressed. In the upcoming section, we will outline a redesigned dashboard that addresses these issues and improves usability, as well as add more notable features that should be presented in the dashboard.

4.2 Redesigned QDAcity dashboard

The new personal dashboard provides users the ability to be able to configure the dashboard themself with the concept of cards. Cards are predefined, such as projects-card, courses-card, newsfeed-card, etc... This allows users to easily add, remove, or change the order of the cards as they want to. However, to give the user an intuitive navigation when first logging in, the dashboard will provide two default important cards for users to work with, the projects card and the courses card.

[1]https://cloud.google.com/
[2]https://cloud.google.com/storage
[3]https://redis.com/

Despite being called customizable, the users may not have access to every attribute of the cards as well as the dashboard. Because there are some features that most QDAcity users don't need and they don't provide much value. For instance, "adding too many colors is one of the most important don'ts in dashboard design" (Stojanovic, 2022), the color-changing functionality is decided not needed for the card, as the color theme for the whole QDAcity software is already defined by a UI/UX team for the best overall user experience when using the software. Adding functionality to change the card's color will not only break the color theme but also reduce the minimalism of the design by overpopulating the dashboard with buttons and colors. Therefore we don't implement the adding color functionality for the cards. Another example of redundant information is the card height, we generally want the card to line up together instead of taking their own lines and leaving a lot of white space, this is another feature that can lead to adverse effects for a customizable dashboard.

4.3 Data modeling

To store the dashboard settings configuration of the users in the database, three data models are being considered. According to the first law of software architecture (N. Ford & M. Richards, 2020) "Everything in software architecture is a trade-off". There is no perfect solution that fits all, we need to decide by doing our trade-off analysis (Magri, 2022). This section examines the advantages and disadvantages of each data model and outlines the ultimate selection and the reasoning behind it.

4.3.1 Dashboard Settings with Cards

Our initial idea of storing the configuration is using both dashboard settings entity and card entities. Each user will have a dashboard settings entity, and each dashboard settings entity will contain multiple card entities, see Figure 4.1

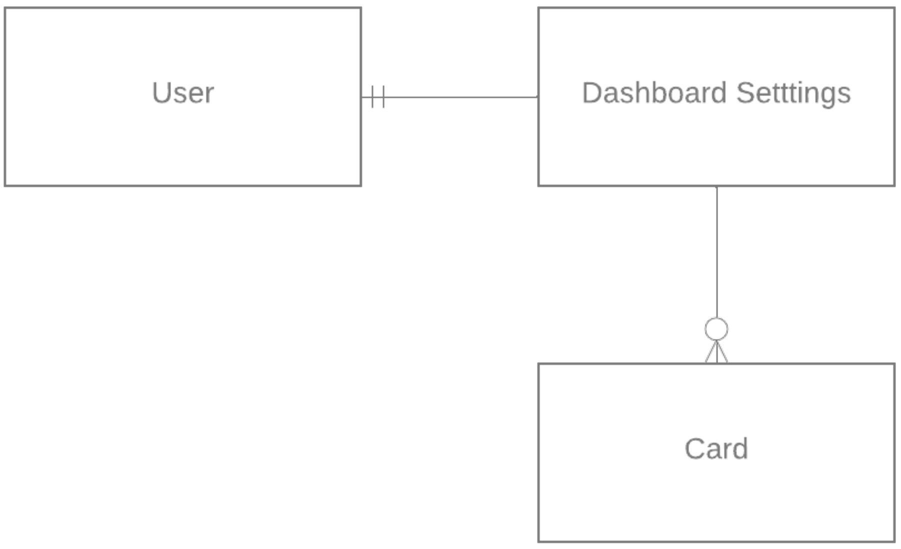


Figure 4.1: Dashboard Settings with Cards

Since each card is independent and the dashboard settings of each user can store as many cards as it needs, this approach has the advantage of storing as much information as we want in each card. Another benefit is that the dashboard settings can be easily extended and can be used not only for the personal dashboard but also the project dashboard and course dashboard by having a list of cards as attributes for each dashboard configuration.

One downside of this approach is that having two separate entities demands more storage in a database. The bigger disadvantage is that having two entities to store information requires more DB requests when loading the dashboard, which increases the loading time when first logging into the new personal dashboard or making changes to the dashboard. One solution to reduce the query time is using the entity groups concept (T. Siu & K. Ardiff, 2019). However, after considering the pros and cons we decided it was not worth using two separate entities. There isn't that much information to store in each dashboard and this will make the implementation unnecessarily more complicated. For the personal dashboard, we only need the card type information and the IDs of pinned projects and courses.

4.3.2 Using only Card Entity to store configuration

Since we don't want to make too many queries, another idea is to eliminate the dashboard settings part and directly point the card to the user, figure 4.2.

Because we don't want to make changes to the user entity to avoid data migration. We will store the owner's value in the card entity instead of making a card list attribute in the user entity, resulting in multiple cards having the same owner.

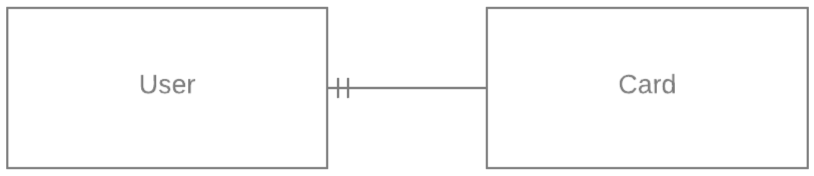


Figure 4.2: Using only card entity to store configuration

The advantage of this is we are still able to store as much information as we want on each card and have faster, simpler queries than the previous data model. We have different card types for different types of information, such as pinned projects, pinned courses, personal state, etc... However, there will be a card that may contain an empty body for their triviality. For example, the news feed card just needs the ID of its owner, once we can get the card type the frontend can easily read the news from the code base without needing to touch the card's information from the backend, it duplicated to have the same piece of card type with just different owner's id. Besides, each user will have several cards, so going through an entire database to retrieve cards corresponding to the user might not be a good idea, as it might slow down the system. We also need extra attributes for the order of the cards, as we don't want to make changes to the user. Making changes to the order of the cards or deleting one of the cards is not ideal because we have to make a lot of changes to not only one but a lot of cards.

4.3.3 Using Dashboard Settings to store configuration

Another model we considered is using the dashboard settings entity to store all the information needed for the user's configuration (figure 4.3). By using the DashboardSettings entities, we trade off the amount of information that each card can hold for query speed. Although some information about each card is limited, we have decided that they are not needed in 4.2. Therefore, here we will select the important information to store in our database, which is about the card order, and list of pinned projects and courses.

Our advantage of using this model is not only avoiding over-complicated implementation but also saving redundant information. For example, the news-feed card will be duplicated with the same information if we implement using only a card entity (from subsection 4.3.2, every user will have one news feed card using that data model). Most importantly the query efficiency is increased when either loading the page or making changes to the page, which play an important role in the user experience (Georgiou, 2014).

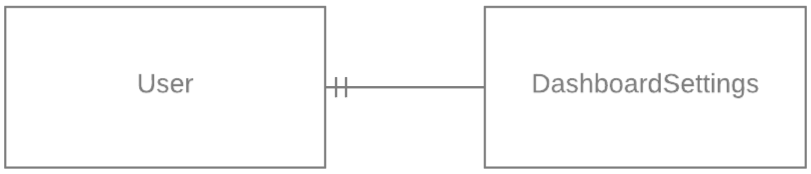


Figure 4.3: Using only Dashboard Settings entity to store configuration

4.4 Data flow in QDAcity

Like the OSI model (J. Day & H. Zimmermann, 1983) as shown in Figure 4.4, we want to divide the networking of QDAcity into layers in order to easily organize the protocols. For the QDAcity dashboard, we divide the system into the following layers, they are Frontend Endpoint, Backend Endpoint, Controller, DAO, domain classes, and Firestore, see Figure 4.5.

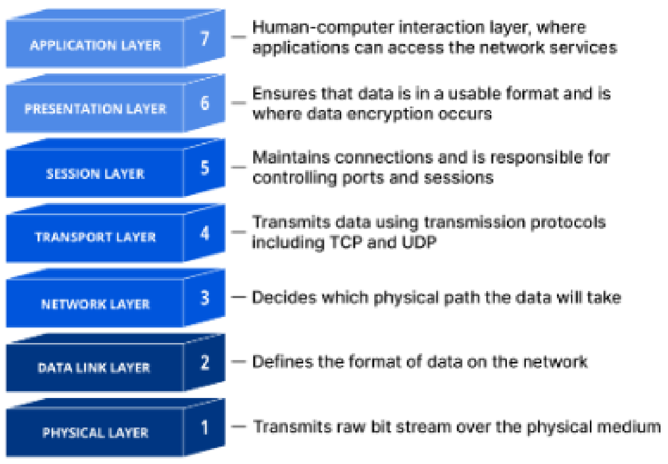

APPLICATION LAYER
7
Human-computer interaction layer, where applications can access the network services
PRESENTATION LAYER
6
Ensures that data is in a usable format and is where data encryption occurs
SESSION LAYER
5
Maintains connections and is responsible for controlling ports and sessions
TRANSPORT LAYER
4
Transmits data using transmission protocols including TCP and UDP
NETWORK LAYER
3
Decides which physical path the data will take
DATA LINK LAYER
2
Defines the format of data on the network
PHYSICAL LAYER
1
Transmits raw bit stream over the physical medium

Figure 4.4: OSI Model

The dashboard on the client side uses the frontend endpoint to communicate with the server side by making API Requests. The backend endpoint is the first layer of the server side, which defines the endpoints of the API. The main goal of the backend endpoint is to offer logic regarding data transport and provide valid routes for the user, here an authentication check for the registered user is provided. Some examples of endpoints here are reading (GET), updating (PUT), or deleting (DELETE). The backend endpoint utilizes the Controller provided by the Controller layer to serve the client. The controller layer is meant to implement business logic and orchestration of DAOs and other controllers. The main goal of the Controller layer is to offer services to the backend endpoint later. The DAO will then take the responsibility to persist the data for the domain classes, create new entities, apply changes, or delete the data from the database. Data persistence and providing services to the Controller is the main goal of this layer. Finally, the data is persisted in Firestore[4] (also known as Datastore), which is a highly scalable NoSQL database running on Google Cloud Platform.

[4]https://cloud.google.com/firestore

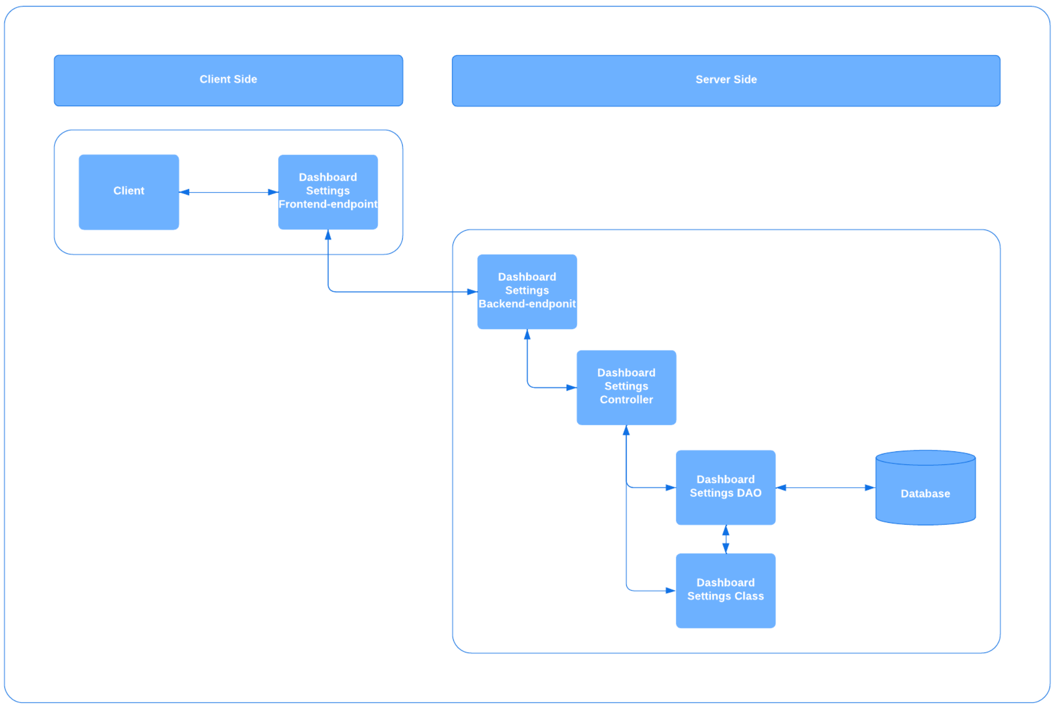

Client Side
Server Side
Client
Dashboard Settings Frontend-endpoint
Dashboard Settings Backend-endponit
Dashboard Settings Controller
Dashboard Settings DAO
Database
Dashboard Settings Class

Figure 4.5: Dashboard Settings Architecture

5 Design and implementation

This chapter describes the details of the design of the implementation of the new dashboard based on the architecture presented in chapter 4. The goal is to fulfill the requirements from chapter 3. We are going to divide the implementation into two parts, the backend part and the frontend part.

5.1 Frontend

In this section, the implementation of the personal dashboard on the client side is described. This involves how the dashboard component is built, its integration with the backend, and the management of data using Javascript with the React library.

5.1.1 Reactjs

Functional components against class components

Starting from React version 16.8, "Anything that can be done using class components can also be done using functional components" (Kong, 2022). Functional components are clearer to read and less complex, with less code than class components. Some features that we could not previously use in class components, for example, the lifecycle hooks are now can be substitution with the *useEffect* hook in functional components. Given that QDAcity is using React 18.2.0, we are going to write new components using functional components.

Prop drilling versus Context API

In order to pass data between components in React, we use something called props, also known as properties. Without using a state management library such as Redux[1] there are two mechanisms to transmit data through the component tree, either manually passing the data through multiple layers of a component hierarchy until the data reaches the desired component, which is prop drilling[2],

[1]https://redux.js.org/

[2]https://dev.to/codeofrelevancy/what-is-prop-drilling-in-react-3kol

or directly passing the data to the components in the tree that needed it with the Context API[3] (see Figure 5.1). Considering that our dashboard architecture in the front contains only three layers, with only the personal dashboard as the parent component and the default card / specialized card using the default card as the children component, and the fact that context API might cause unnecessary re-rendering in Application, it's decided better, in this case, to use the drilling strategy to passing card's props through the component tree. For props that are passed down from higher or the root component (the *App.jsx*) we are going to use the provided context hook (for example *useAuth* hook) to keep our code clean.

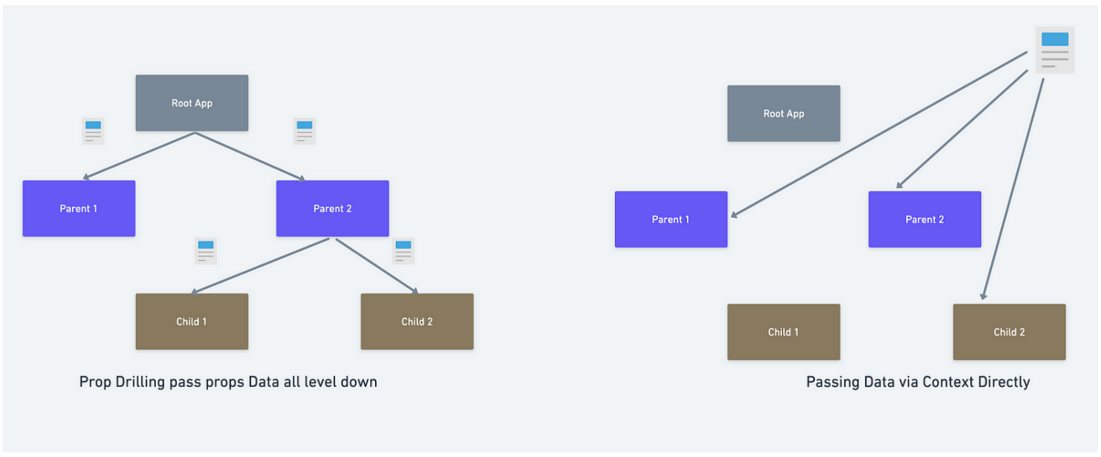

Root App
Parent 1
Parent 2
Child 1
Child 2
Prop Drilling pass props Data all level down
Root App
Parent 1
Parent 2
Child 1
Child 2
Passing Data via Context Directly

Figure 5.1: Prop drilling vs Context API

Styling

For our styling, we are going to use the styled component library[4], which is a CSS in JS writing technique. Writing actual CSS in react components keeps the flexibility to make changes or reuse the CSS code, as well as makes code easier to read and maintain. Some features of styled components over traditional CSS classes are simpler extending, nesting styles, and especially dynamic styling, as written in the motivation on the library's website "Adapting the styling of a component based on its props or a global theme is simple and intuitive without having to manually manage dozens of classes"[5]. By making use of that we can have a consistent color theme, which is defined in a common file in the project, passing down the props theme color and we are ready to use, we can see in example 5.1. This also makes implementing of dashboard and card a lot easier, as we can reuse a lot of styling code, such as button components. In this thesis, we are trying to use as much reusable code as possible to help maintain the project later on.

[3]https://react.dev/reference/react/useContext
[4]https://styled-components.com/
[5]https://styled-components.com/docs/basics#motivation

```
const StyledCardContainer = styled.div`
  background-color: ${(props) => (props.isDragOver ?
     props.theme.bgSecondaryHover : props.theme.bgDefault)};
  height: ${CommonDimensions.card.heightPx}px;
  width: ${(props) => props.cardWidth}
  padding: 16px;
  & > h1 {
    font-size: 24px;
    font-weight: bold;
  }
  & > p {
    font-size: 16px;
  }
`;
```

Code Example 5.1: Card Style Container

5.1.2 Card and Dashboard implementation

Desktop first design

Most QDA software users use the software on a desktop, and the product is intended for desktop use. We will start our design based on a desktop computer, hence start writing CSS for large viewport sizes first. Then we use CSS media queries to alter the experience for smaller ones.

FormatMessage, Button, ToolTip

We have several rules for the front-end conventions. The first one is all the colors must be defined in the *Theme.js* file. Subsequently, in QDAcity there are two main languages being used, English by default and German. This is why every user-facing string has to be formatted so that we can serve them both in English and German. We use the React integration of *Format.js* API[6] called *React-Intl* with *FormatedMessage* component (see code example 5.2) to implement that.

```
<StyledEditButtonContainer>
    <PrimaryButton
        id="EditDashboardButton"
        onClick={handleEditClick}
        label={
```

[6]https://formatjs.io/docs/react-intl/

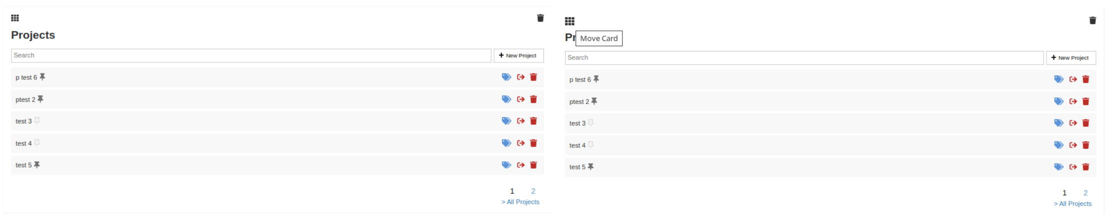
```
            isEditing ? (
                <FormattedMessage id="personalDashboard.saveDashboard"
                    defaultMessage="Save Dashboard" />
            ) : (
                <FormattedMessage id="personalDashboard.editDashboard"
                    defaultMessage="Edit Dashboard" />
            )
        }
    ></PrimaryButton>
</StyledEditButtonContainer>
```

Code Example 5.2: Button with FormattedMessage as label

As for the buttons, if the buttons contained text, they would have to follow the above convention. For buttons that are only icons or the usage of the buttons is not clear, tooltips are needed for more information about it, these tooltip has to follow the format convention as well. An exception for the tooltip is if the button already has text to explain itself and is too simple that they don't need a tooltip to provide more information about it (for example the edit dashboard button).

Card

Our card implementation comes with three different types of default card templates. These default cards are only different in their container, which defines the width of those cards. These cards' names are *DefaultCard* which will take a width in pixels as props for their width, by default the same size as the small card. The *BigCard* and *SmallCard* widths are defined in two different screen sizes to keep them responsive to the *MediaSize.js* used for screen size breakpoints and *CommonDimensions* used for the common width files defined in common assets of the QDAcity codebase. In medium screen size and more, the big card will take about half the width of its parent component's width while the small card will take about one-quarter. The reason for those approximate laying is to fit the card on the screen with the distance of the gap between each card, which is designed to be responsive with percentage value also. These cards have their *isEditMode* and *isDragOver* state, passed down from the personal dashboard to handle their visual. If the *isEditMode* condition is true, two icons are being shown for deleting and moving the cards around. We also have the hovering effect (icon gets bigger or changes color, pagination has underline) to help the users interact with the buttons (figure 5.2).



(a) Move icon not hovered **(b)** Move icon hovered with tooltip

Figure 5.2: Projects card in edit mode

The three cards stated above work only as a holder without any content. To fully create a card, we wrap one of the default cards with its title and contents, then pass it down to the default card with children-prop. The default card will then take the children's element and generate it inside the *StyledCardContainer*. After this step, we can import our desired card and render it in the personal dashboard as wanted. (Code example 5.3).

```
<PersonalDashboard>
    ...
    /// props from Personal Dashboard
    <ProjectsCards>
        ...
        /// props passed down from Personal Dashboard
        <BigCard>
            ....
            /// props passed down from Personal Dashboard and the
                children of Projects Card
            {children}
        </BigCard>
        ...
    ></ProjectsCards>
    ...
</PersonalDashboard>
```

Code Example 5.3: Props passed down from Personal dashboard

Dashboards

To manage the cards in the personal dashboard, we are going to use a state hook to keep track of the user interface changes. We implement the change card order for the dashboard by creating drag and drop functionality. In order to move a card, the dashboard must be in edit mode and the user has to move the card while holding the move card button. The steps for moving the cards are as follows, the button being clicked, triggering the *handleDragStart* function, hovering it in or out a card triggers the hover drag functions in the card. When the user releases the mouse, the *handleDrop* function is triggered, changing the index of the card using the splice function. Although objects in React state are technically mutable, directly modifying the state array can lead to unexpected behavior, as recommended in the React document we should treat React objects as if they were immutable. So we will first need to make a copy of the state, make changes to it, and replace the original with the copy.

To synchronize changes to the backend we use *useEffect*[7] hooks in two ways. The one initial hook we use is to initialize our dashboard settings when the dashboard is first loaded, there for the hook dependency is set empty. The other hooks are for updating the changes in the card order, so we will set the dependency to the card's state. However, the *useEffect* hook for updating the card is called when first rendered as well, which will catch a conflict with the first initializing hook. We will avoid that situation by adding a *useRef*[8] hook with a boolean value, which is primarily used to access and manipulate the DOM or to store mutable values that don't trigger re-renders. If the condition is true we will set it to false and return the update hook early. After that, the update *useEffect* hook should work as desired and won't cause any conflict with the first hook.

Navigation

We use *useNavigate* hook[9], a hook that is introduced in React Router v6 to navigate. This hook is used together with the *HistoryAndLocationProvider* to create a history API to go to specific URLs and forward or backward pages (code example 5.4).

```
const navigate = useNavigate();
const getNextUrl = useCallBack((url) => {
    ...
}, []);
const history = {
    push: (url, state) => {
        return navigate(getNextUrl(url), { state });
    },
    replace: (url, state) => {
        return navigate(getNextUrl(url), { state, replace: true });
    },
    ...
};
```

Code Example 5.4: History and location provider

[7]https://react.dev/reference/react/useEffect
[8]https://react.dev/reference/react/useRef
[9]https://reactrouter.com/en/main/hooks/use-navigate

In some cards, such as *NewsFeedCard*, only one new is displayed. In order to get all the changes of QDAcity, we are going to add a link to the bottom right of the card and use the function *history.push('/latest-changes')* to navigate to the news feed page. We also provide a navigation menu for the user to go back to the previous page easily.

Filter

For filtering pinned projects in the projects list view we pass down a boolean *doShowOnlyPinnedProjects* props from the parent component to *ProjectList*. The project items will be filtered and then passed down to *ItemList* for item rendering (code example 5.5).

```
const isProjectPinned = (project) => {
    return pinnedProjects && pinnedProjects.includes(project.id);
};
<ItemList
   ref={(r) => {
      if (r && !itemList) setItemList(r);
   }}
   hasSearch={true}
   hasPagination={true}
   doNotrenderSearch={true}
   itemsPerPage={5}
   items={
      doShowOnlyPinnedProjects
         ? projectsContext.projects.filter((project) =>
            isProjectPinned(project))
         : projectsContext.projects
   }
   renderItem={renderProject}
/>
```

Code Example 5.5: Project items filter

5.2 Backend

This section describes the implementation of the dashboard settings on the server side, including the data model, the endpoint, controller, and DAO implementation in Java programming language.

5.2.1 DashboardSettings data model

First, a data model for storing the configuration of the user needs to be created, using the model we discussed in section 4.3.3. The dashboard settings entity with an overview of its attributes and relationships are defined as shown in figure 5.3.

The entities from our data model are stored in the GAE datastore, which is a key-value datastore that is conceptually similar to a HashMap. The operations in the datastore are handled by a Java library called Objectify[10], this will be discussed in more detail in sub-section 5.2.2. Here we are focusing more on the properties model and their annotations of the data model. In Objectify, the *DashboardSettings* class is marked as an entity with class-annotation *@Entity*. After that, we define our ID with *@Id* annotation. The rest of our attributes are annotated with *@Index* to specify that they should be indexed in the data store. A snapshot of the code is shown below in code example 5.6.

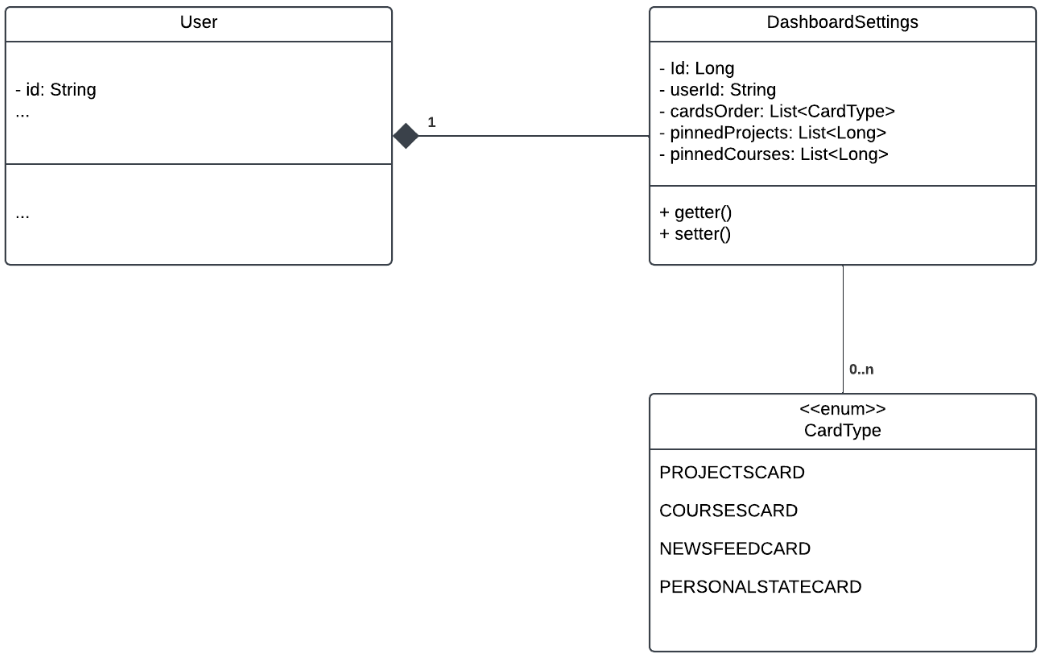


Figure 5.3: DashboardSettings Data model

```
@Entity
public class DashboardSettings implements Serializable{
    @Id
    Long id;
```

[10]https://github.com/objectify/objectify/wiki

```
    @Index
    String userId;

    @Index
    List<CardType> cardsOrder;

    @Index
    List<Long> pinnedProjects;

    @Index
    List<Long> pinnedCourses;
```

Code Example 5.6: Class and attributes annotations

5.2.2 DashboardSettingsDAO

According to Martin (Martin, 1983), create, read, update, and delete (CRUD) are four basic operations of persistent storage. Our DAO design pattern also provides these four operations with a Java data access API called Objectify. This pattern handles all the DB-interface-specific code, which will keep the usage of the database not scattered throughout the code base, keeping the code clean and easy to refactor/reuse later. Although using different terminology, Objectify is implemented with these basic operations. In *DashboardSettingsDAO* we also use the *now* call to extract value from the asynchronous result and return it to the *DashboardSettingsController* (except for the delete method which returns value is null). Another thing to acknowledge here is when creating a *DashboardSettings-DAO* object, we are not going to directly call the constructor but we are using a Static Factory Methods (Block, 2009), this helps the code easier to read as we can call the name of the class with the *with* method to add context parameter. An example of code is shown below in code-example 5.7.

```
    public static DashboardSettingsDAO with(Context context) {
      return new DashboardSettingsDAO(context);
   }

public DashboardSettings getDashboardSettingsForUser(String userId) {
      Query<DashboardSettings> query =
          ObjectifyService.ofy().load().type(DashboardSettings.class);
      query = query.filter("userId", userId);
      return query.first().now();
   }
```

Code Example 5.7: getDashboardSettingsForUser method in DAO class

5.2.3 DashboardController

Within the *DashboardController* the business logic is managed. While the update calls simply pass down the updated dashboard to the DAO and the delete call just calls the DAO method, the get method has to check the availability of the dashboard settings when users first log in. It will have to call the initialize method if there is no dashboard for that user and return that dashboard setting for the next sequence calls of the get methods. For initializing we added the two most basic cards for the user to start with QDAcity coding, the projects card, and the courses card. Again we are using Static Factory Methods for creating *DashboardController* instead of directly calling the Constructor. Code examples 5.8 of the *initDashboardSettingsMethod*. When removing a user, the *deleteDashboardSettingsForUser* method is called, delete the dashboard before removing the user.

```
public DashboardSettings initDashboardSettings(String userId, User
    user) {
   DashboardSettings dashboardSettings = new DashboardSettings();
   List<CardType> initialCardsOrder = new ArrayList<>();
   initialCardsOrder.add(CardType.PROJECTSCARD);
   initialCardsOrder.add(CardType.COURSESCARD);
   dashboardSettings.setUserId(userId);
   dashboardSettings.setCardsOrder(initialCardsOrder);
   return
       dashboardSettingsDAO.insertDashboardSettings(dashboardSettings);
}
```

Code Example 5.8: initDashboardSettings method in Controller class

5.2.4 DashboardEndpoint Backend

The *DashboardEndpoint*, using GCE in the backend works as an authorization layer, defines the REST endpoint for the frontend, and transmits data between the frontend and the business layer. The *@Api* annotates for API-wide configuration and *@ApiMethod* marks a method as an Endpoint with the function name to be called at the front end, the HTTP method, and the web path parameters. An example of code 5.9 is shown below.

The *Context.executeWith* method has two parameters, a *User* parameter, and a *ContextExecuteable*. This method will check for the authentication information of the user and throw an *UnauthenticatedException* if the user is not authenticated. If the user is authenticated it will then execute the given unit of work.

```
@Api(
    name = "qdacity",
    version = Constants.VERSION,
    namespace = @ApiNamespace(
      ownerDomain = "qdacity.com",
      ownerName = "qdacity.com",
      packagePath = "server.project"),
   authenticators = {QdacityAuthenticator.class}
)
public class DashboardSettingsEndpoint {

   @ApiMethod(name = "dashboardSettings.initDashboardSettings",
      httpMethod = "PUT",
      path= "dashboardSettings/users")
   public DashboardSettings initDashboardSettings(@Named("userId")
       String userId, User user) throws UnauthorizedException {
      return Context.executeWith(user, context -> {
         return DashboardSettingsController.with(context)
                   .initDashboardSettings(userId, user);
      });
   }
```

Code Example 5.9: Dashboard Settings Endpoing example

6 Evaluation

In this chapter, we evaluate whether the presented FR and NFR from Chapter 3 are fulfilled.

6.1 Functional Requirements

FR-1 If the user logs in for the first time, the personal dashboard shall provide two default projects and course cards to the personal dashboard.

When logging in for the first time, the *initDashboardSettings* method is called and a *DashboardSettings* with a default cards order containing projects card and courses card is created for the user.

✓The requirement FR-1 has been fulfilled.

FR-2 The personal dashboard shall provide users with the ability to save their dashboard configuration.

With the implementation of two *useEffect* hooks, one with *cardsOrder* and another with *pinnedProjects* dependency, the *updateCardsOrder* and the *updatePinnedProjects* APIs are called every time the cards order or pinned projects changes and saves the dashboard configuration.

✓The requirement FR-2 has been fulfilled.

FR-3 As soon as users log in to QDA, the personal dashboard shall provide users with the ability to retrieve the dashboard configuration they saved before.

With the implementation of the *useEffect* hook with empty dependency, the *getDashboardSettingsForUser* API is called and returns the dashboard configuration users saved before.

✓The requirement FR-3 has been fulfilled.

FR-4 If the users enter edit mode, the dashboard shall provide the users with the ability to add/remove/move cards.

By implementing these buttons with the *isEditmode* state as in subsection 5.1.2, these buttons appear when the user enters edit mode and can be interacted with.

✓The requirement FR-4 has been fulfilled.

FR-5 The projects list view shall be able to present all the projects.

The projects list view gets all projects that belong to a user and presents them in the list view.

✓The requirement FR-5 has been fulfilled.

FR-6 If the user wants to have a view of only pinned projects, the projects list view should be able to filter only pinned projects.

A pinned projects filter has been implemented in the form of a checkbox for the user.

✓The requirement FR-6 has been fulfilled.

FR-7 If there are projects that the user frequently uses, these projects shall be able to be pinned to the pinned projects card in the personal dashboard.

Pinned icons are added to each project in the projects list view.

By implementing the *pinnedProjects* attribute and adding a pin icon to the project, the projects can be pinned to the pinned projects card in the personal dashboard.

✓The requirement FR-7 has been fulfilled.

FR-8 If the users want to navigate between the personal dashboard, projects list view, and project dashboard, the dashboard system shall provide the user a clickable link to navigate between them.

In the personal dashboard user can click on > *All Projects* link to navigate to the list view. A link at the top of the projects list view and a home icon that redirects to the personal dashboard in the navigation menu is provided.

✓The requirement FR-8 has been fulfilled.

FR-9 If the users follow a course/project, they are able to turn on the notification in customized dashboard mode by clicking on the alarm icon, so that they will receive news about that course/project.

We haven't implemented this due to time constraints and lower priority.

X The requirement FR-9 has not been fulfilled.

FR-10 If the project is pinned by a user, the database shall be able to store the pinned project information only for the user who pinned it.

With the implementation of the *DashboardSettings* entity with only one user ID attribute and the *updatePinnedProjects* method call only update the *DashboardSettings* of that user, only the user who owns that setting can see the pinned projects.

✓The requirement FR-10 has been fulfilled.

FR-11 If the pinned project is deleted, the database shall delete the information about the pinned project.

We have added the *updatePinnedProjects* API call to change the list of pinned projects when a pinned project is deleted.

✓The requirement FR-11 has been fulfilled.

FR-12 If the user profile is deleted, the dashboard settings profile shall be deleted.

The *removeUser* function now also deletes the dashboard settings of its user before deleting that user.

✓The requirement FR-12 has been fulfilled.

6.2 Nonfunctional Requirements

NFR-1 The dashboard should be designed in a way that can be used easily by the users without the need for any tutorials.

The dashboard's card has its own information and the buttons are designed with minimalism, a lot of space, intuitive icons, tooltip for extra information.

✓The requirement NFR-1 has been fulfilled.

NFR-2 The dashboard should be designed in a way that allows users to quickly access projects/courses and interpret relevant information to the project/-courses such as their titles or user's role in that project/course.

With the projects card in the personal dashboard, the user can simply navigate to the project with just a click.

✓The requirement NFR-2 has been fulfilled.

NFR-3 The buttons in the dashboard should be designed in a way that interacts with users and helps people use their functionality by providing a tooltip.

The tooltips are implemented in both languages for the user.

✓The requirement NFR-3 has been fulfilled.

NFR-4 The components and styling should be designed in a way that is compact for different web browsers, such as Google Chrome, Mozilla Firefox, and Brave.

The dashboard is manually tested using Microsoft Edge, Chromium, Google Chrome, Mozilla Firefox, Brave and works for all of them.

✓The requirement NFR-4 has been fulfilled.

NFR-5 The color scheme should be designed in a way that matches the color scheme of QDAcity.

During the implementation of the dashboards, we reused a lot of components, such as buttons, item lists, and icons. Additionally, all the colors we used are defined in a Theme.js file.

✓The requirement NFR-5 has been fulfilled.

NFR-6 The dashboard settings of each user shall be designed in a way that can only be modified by its owner, which means the cards can be added/deleted/positioned only by the user who created them.

Each API call checks for the user authorization of the owner, only the owner of the dashboard can modify it. If another user tries to make changes to the other dashboard, an *UnauthorizedException* is thrown.

✓The requirement NFR-6 has been fulfilled.

NFR-7 The dashboard should be designed in a way that is easily reusable for project and course dashboards.

We have the card concept developed and can be easily reused on another dashboard, we just need to wrap them around the list item element.

✓The requirement NFR-7 has been fulfilled.

NFR-8 QDA Dashboard should be designed in a way that gives users a good user experience, including the intuitiveness of the dashboard, ease of navigation between projects, and the ability to quickly access relevant information.

All important information is centralized, navigation menu is created. However, the move card functionality works when dropping one card on top of another. When moving a card between two cards, the function is not working, which is not intuitive when managing cards for users.

✓X The requirement NFR-8 has been partially fulfilled.

NFR-9 The Dashboard must be designed in a way that will adjust responsively to the width of other devices outside of a computer such as a phone or tablet.

All the components in the dashboard are designed with the query check for media size and using percentages to adapt to make the dashboard more responsive.

✓The requirement NFR-9 has been fulfilled.

7 Conclusions

The last chapter concludes the work and sums up the thesis. The goal of this thesis is written down in Chapter 1, motivated and set the target for this thesis.

Chapter 2 summarizes our related work and research for this thesis. Firstly we introduced QDAcity software, its component architecture, and its most important features. Subsequently, we discussed how the old implementation of the dashboard could affect the quality of the software as well as the user experience. To that, we introduced our new way of designing and implementing a dashboard that can give users the best experience and a quick look at the navigation structure of the new dashboard.

In Chapter 3 we formulate functional requirements and non-functional requirements of the dashboard. As the thesis focuses on the user experience, our most important requirements will be the intuition of the dashboard, and how it should be easy to navigate and use.

Following is the architecture of the dashboard in Chapter 4. Here we had a quick brief about the old architecture and came up with the new one. We talked about how the data flows from the client to the server. Then we discussed options for a data model for the dashboard settings and gave our ultimate decision.

Chapter 5 describes our implementation of the architecture that we had in Chapter 4. Here we had a details description of every component we had for the dashboard, as well as reasoning for our choices to approach the problems. For each component that we had, a code example is provided for a better illustration of our work.

In Chapter 6 we revisited and evaluated the fulfillment of our work regarding the requirements we stated in Chapter 3. There we had our most significant requirements fulfilled.

To conclude the thesis, we have satisfied the implementation of a customizable dashboard for QDAcity. Giving QDAcity a new look and providing a better experience for the user.

References

Abbas, T. (2023). Minimalism in graphic design: Top trends to watch in 2023. https://www.linkedin.com/pulse/minimalism-graphic-design-top-trends-watch-2023-toqeer-abbas/

Anderson, S. (2023). How fast should a website load in 2023? https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/#:~:text=following%20key%20findings%3A-,47%20percent%20of%20consumers%20expect%20a%20web%20page%20to%20load,render%20before%20abandoning%20the%20site.

Block, J. (2009). *Effective java.*

Georgiou, M. (2014). Need for speed – fast loading the key to a satisfying ux. https://www.getfeedback.com/resources/ux/need-speed-fast-loading-key-satisfying-ux/

J. Day & H. Zimmermann. (1983). *The osi reference model.* IEEE.

Kong, L. (2022). React component guide: Class vs functional. https://www.educative.io/blog/react-component-class-vs-functional

Lygenda, D. (2022). Single page application vs. progressive web app: A comparison. https://www.microverse.org/blog/single-page-application-vs-progressive-web-apps-a-comparison

Magri, B. (2022). [summary — chap 2] fundamentals of software architecture. https://medium.com/@biancamagri/summary-chap-2-fundamentals-of-software-architecture-ec4532901285

Malak, A. (2022). What is information overload? how to overcome it? https://theecmconsultant.com/information-overload/

Martin, J. (1983). Managing the data-base environment. Englewood Cliffs, New Jersey: Prentice-Hall.

Miller, G. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. https://psychclassics.yorku.ca/Miller/

N. Ford & M. Richards. (2020). *Fundamentals of software architecture: An engineering approach.* O'Reilly Media.

Shah, A. (2014). The 3 clicks rule how many clicks should it take to reach your content? https://medium.com/@allyshah_design/the-3-clicks-rule-c9bb5eaf7d31

Sisense. (2023). Dashboard design best practices - 4 key principles. https://www.sisense.com/blog/4-design-principles-creating-better-dashboards/

SOPHIST. (2016). Schablonen für alle fälle. https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/Wissen_for_free/MASTeR_Broschuere_3-Auflage_interaktiv.pdf

Stojanovic, F. (2022). Bad dashboard examples: 10 common dashboard design mistakes to avoid. https://databox.com/bad-dashboard-examples

T. Siu & K. Ardiff. (2019). Entity groups, ancestors, and indexes in datastore- a working example. https://medium.com/google-cloud/entity-groups-ancestors-and-indexes-in-datastore-a-working-example-3ee40cc185ee

Tidwell, J. (2005). Designing interfaces. https://www.oreilly.com/library/view/designing-interfaces/0596008031/ch04.html

Walker, T. (2022). Why simple website design is the best: The scientific reasons. https://cxl.com/blog/why-simple-websites-are-scientifically-better/