Wrocław University of Science and Technology

Faculty of Information and Telecommunication Technology

Field: ISTANG

TEAM PROJECT

Software as a Service Solution for Mobile Management of IoT Devices

Agata Piasecka

Justyna Dul

Szymon Misztal

Przemysław Świat

Supervisor

Dr. hab. Krzysztof Juszczyszyn

Keywords: Internet of Things, Mobile, SaaS

WROCŁAW (2023)

DESIGN DOCUMENTATION

1. List of symbols, designations and acronyms

FR - functional requirement
NFR - non-functional requirement

2. Purpose and scope of the project

General objective

The platform empowers users to design sophisticated, cohesive interfaces that streamline management, data visualization, and seamless integration of their IoT devices. It fosters collaborative development among teams in shared workspaces, ensuring utmost security and data integrity. Accessible remotely, all these capabilities are conveniently available on the user's mobile phone.

Problem statement

IoT Devices nowadays require a flexible, remote solution for management and configuration from any proximity for an easier and more efficient development process and operation [1]. Having stationary equipment forces the Users and developers to relocate the IoT Device for real-time observation and manipulation, which may be very costly.

Users want to create, and personalize mobile phone interfaces to facilitate management and data representation for their IoT Devices. They also want to integrate their devices with external mobile/web services, and automate processes [2]. This process may prove time consuming and challenging without appropriate software.

Solution

The system we offer is a software-as-a-service solution that provides users with tools to create interfaces from mobile to IoT devices and integrate their devices with external web/mobile services.

The system simplifies the development and usage processes by offering a secure, remote solution with complex functionalities, enabling the manipulation of IoT devices. It includes comprehensive data representation and management tools, along with enhanced collaboration tools.

An integrated mobile solution allows convenient access to IoT devices from any proximity, offering complex configuration and management capabilities for any IoT device with suitable interfaces. Collaboration tools further streamline the development process and management of IoT devices.

Scope

The Mobile App will be initially distributed on the Android platform, with the IOS version and mobile web app being a possibility in the future.

3. Glossary

All the terms of this glossary will be mentioned in the document with the first letter capitalized.

System
All the applications developed by our team. It includes both the System's Backend Server and the System's Mobile App.

Backend Server
Backend application developed by our team as part of the System with all its dependencies.

Mobile App
Android mobile application developed by our team as part of the System.

User
Person that interacts with any part of the System.

Admin
User that has special privileges in the System, such as managing Users.

Message
The smallest unit of data used in communication between the System and the IoT Devices.

Topic
Data storage for Messages. The System and the IoT Devices can both read from and write to the Topics.

IoT device
Programmable device interacting with the System via Topics.

Component
The smallest unit of widgets used for interaction between IoT Devices and the System. It is managed by the Users. The interaction occurs with the use of Topics.

Dashboard
A group of Components within one Mobile App's screen, created and shared by a group of Users. Dashboard can be edited only by Users with access privileges. The Components within the Dashboard operate on a specified group of topics.

Project
Group of Topics and Dashboards managed by its owner. It is available to Users with granted access privileges.

Project Owner
User that created the Project.

Project Admin
User managing the Project's members.

Project Editor
User with editing capabilities for the given Project.

Project Reader
User that can view and interact with the Project but not modify its structure.

Input Component
A Component used for sending Messages from the Mobile App to the IoT Devices via Topics.

Output Component
A Component used for the representation of data coming from IoT Devices via Topics.

Trigger Component
A Component that triggers an action in response to a specified event. Actions and events can involve either IoT Devices, or mobile/web services integrated with the System.

4. State of knowledge in the project field

Overview

There are some similar solutions already on the market. In this section, we will shortly describe each of these products, and compare them in terms of functionalities, and designs.

Alternative solutions

Home Assistant

Home Assistant is an open-source home automation platform that allows users to control and manage various smart devices and services within their homes. It serves as a central hub for connecting and automating smart lights, thermostats, security cameras, and other IoT (Internet of Things) devices.
Home Assistant offers a user-friendly interface, extensive compatibility with a wide range of brands and protocols, and the ability to create customized automation routines, making it a versatile and powerful tool for creating a smart, interconnected home environment [3].

Similarities
- Creating dashboards and interactive blocks
- Creating automations
- Integrations with external web services

- Mobile application

Differences
- No mobile integrations
- Locally hosted
- No real-time collaboration on dashboards
- Harder to integrate devices with no OS
- No inbuilt MQTT server

Adafruit.IO

Adafruit IO is a cloud platform that allows you to create interfaces for your IoT systems on a modern web page. You can create dashboards, components, and link them with IoT devices via MQTT queues. The communication between the devices and the service is conducted through feeds. There are many types of components, such as buttons, sliders, or graphs. You can use them either to send messages to feeds, or display data from feeds. It is possible to integrate your feeds (and consequently your IoT devices) with external web services only via IFTTT.com [4][5].

Similarities
- Communication between the service and the devices via MQTT queues
- Creating dashboards and interactive blocks
- Software as a Service

Differences
- No mobile application
- Integrations with external web services only via IFTTT.com
- No integrations with mobile applications
- There are no tools for collaboration on dashboards

IFTTT

IFTTT (If This Then That) is a web-based automation platform that allows users to create and customize workflows, known as applets, to connect and automate various online services and devices. It enables users to create conditional statements where a trigger event in one service (the "if this" part) can result in an action in another service (the "then that" part). This platform simplifies the process of automating tasks and integrating different online tools and apps to streamline personal and professional workflows [6].

Similarities
- Creating automations
- Integrating external web services with each other
- Software as a Service

Differences
- Cannot be used for control or data display of the IoT devices
- No dashboards and interactive blocks

Innovation

The project greatly expands the similar solutions in the field, adding new functionality together with a mobile interface. It is a useful and flexible solution that can be advantageous to potential users both on the individual and company scales. The added team collaboration feature greatly improves and accelerates the development process of IoT devices. In general, the project creates a new direction for the IoT field, and encourages Users to expand it with their own projects.

Summary

No alternative product provides all the functionality that this solution offers. It is especially distinguished in areas of mobile integrations, and real-time collaboration between many users.

5. Preliminary assumptions

Context

Project context can be used to determine what processes are necessary to finish the project. Its definition may highlight problems for the project in early stages of development. We used a six-dimensional framework described as *Who, Where, What, When, How and Why* to define our project context [7].

- Who (culture): The development team is small, and has knowledge necessary to achieve the goals of the project.
- Where (space and time): The development team is located in the same city, and has regular meetings. The potential users are located anywhere in the world.
- What (product constraints): The product is a software-as-a-service solution available through the mobile application interface. Top quality is expected, as any errors, or downtimes would affect the wide range of different devices, and applications.
- When (product life-cycle): The project development is conducted within 5 months starting on 07.2023. After release, the product shall continue operation, and be available for installation for an indeterminate period of time.
- How (engagement constraints): The potential users are IoT developers, and smart home enthusiasts with some technical knowledge. They will be engaged by user testing during the development.
- Why (organizational drivers): The full scope of the project is expected to be finished within four months of development (reduce time-to-market strategy).

High-level goals

The project is aiming to provide:

- A way to develop mobile phone interfaces for IoT devices.
- Software-as-a-service solutions available through a mobile.
- Simple, unified communication protocol to connect IoT devices with the System.

- Predefined sets of components, inter-application integrations (web and mobile), and automations to design dashboards that can be used to interact with the IoT devices.
- Real-time collaboration within groups and organizations.

Users

We can distinguish three potential groups of Users:

- Developers - IT professionals developing IoT devices and applications. They are going to integrate their devices with the System.
- Smart home enthusiasts - amateurs with technical knowledge that may use the System to create interfaces for their own IoT devices.
- Regular smart home users - non-technical users that will use the System to manage their devices.

Internal stakeholders

People affiliated with the company or project which is their source of employment, investment or possession. Can have direct influence over decision making and key responsibilities.

Przemysław Świat
Team leader and backend developer
Manages and leads team, develops back-end server, fixes bugs, writes tests

Agata Piasecka
Android developer and Agile project manager
Develops android application, designs UI, makes API integrations, fixes bugs, writes test

Szymon Misztal
Android developer
Develops android application, makes API integrations, fixes bugs, writes test

Justyna Dul
Backend developer
Creates new functionalities, fixes bugs, writes tests, solves algorithmic and business problems

External stakeholders

People or organizations not affiliated, that have interest in supporting businesses or projects. Have indirect influence over decision making and directions of development [8].

Home users
User
They install the Mobile App on their phones and integrate the System into their IoT ecosystems. They have to learn how to operate it from the documentation or external sources.

IT companies
User
They integrate the System into their commercial products. The continuity of operation of the System is very important to them.

Government
Legislator, law enforcer
The government checks if the System, its usage, and data collection practices do not break any laws. It can also pass new laws that could affect the System's operations.

Technologies

Choosing proper programming languages and frameworks is key to developing high-quality, robust, and easily maintainable software. It can lower labor costs and development time and this affects the price of the product and its competitiveness [9]. We selected one of the most popular and known to us solutions.

Jetpack Compose - *UI toolkit for creating Android applications* [10].

Reason:

- Modern, highly standardized.
- It simplifies and accelerates UI development on Android.
- Allows for better code reusability.
- Near no boilerplate code.

Spring Boot 3 - *Open-source framework for building Java/Kotlin web applications* [11].

Reason:

- It enables us to make code that is modular, extensible, and easy to maintain.
- There are also a lot of ready-to-use libraries developed for Spring, like JPA, Spring Security, or Spring AMQP, which facilitates the development process.
- A really popular choice for developing back-end servers.
- Possible to be programmed in a very well known to us language, Kotlin.

Kotlin - *JVM-based open-source programming language used for cross-platform projects* [12].

Reason:

- It includes features allowing for skipping unnecessary boilerplate code that occurs commonly in Java.

- Can be used in both, server and mobile app development.
- For mobile apps, it is the most common and best option
- It can use Java libraries

PostgreSQL - *open-source relational database management system* [13]

Reason:

- High data consistency and integrity
- Very flexible and failure resistant.
- It is capable of supporting a range of different applications, most importantly the services handling large numbers of concurrent requests.
- Scalable

Docker - *a platform for distributing packaged applications in the form of containers* [14].

Reason:

- It supports an easy way to include dependencies.
- Easy sharing and publicizing of images.
- It is consistent across various environments.
- Allows for efficient deployment, management, and running software.

RabbitMQ - *open-source message broker software for communication between various components in distributed applications* [15].

Reason:

- It can handle many protocols like MQTT, our main choice for this project.
- It also supports handling queuing, routing, and load balancing.
- Small request-response time
- No constraints for payload size
- Used for simple use cases

Limitations

The decision to develop an Android mobile application for minimal API 26 (Android 8 Oreo) limits our potential users. It is estimated that 6% of devices operate on systems below our expectations, which potentially excludes users who have not updated their system yet.

Docker chosen for more efficient deployment and management of the Backend Server requires supported and enabled virtualization. In environments where virtualization is not feasible or supported, our solution will not work.

Additionally, the Backend Server cannot operate with too many concurrent users which we estimate is around a few thousand.

6. Requirement specification and analysis

Requirement specification is a crucial step in an IT project. It may determine whether the project will be successful or not. We defined user (functional) requirements in the form of user stories, as it is the most popular approach in agile projects [16].

To further concretize the functional requirements, we created use case diagrams, and the use case specification. The use case diagrams illustrate the names of actors, use cases, and their relationships, while specification describes the interactions between Users and the System in detail [17].

Except for functional requirements, we provided non-functional requirements, also called quality requirements. They give more overall details to the requirement specification. They are defined in the form of user stories with acceptance criteria. Each quality requirement is related to many functional requirements [18].

User stories (functional requirements)

1. As a User, I want to exchange data between the System and my IoT Devices remotely, so that I can interact with my IoT Device regardless of my location.
2. As a User, I want to create and manage personalized Dashboards for sets of IoT Devices, so that I can have easy and systemized access to Components that will be used often.
3. As a User, I want to define and manage Components on my Dashboards, so that I can monitor, represent, and exchange data between the Mobile App and my IoT Devices.
4. As a User, I want to define Topics, so that I can create communication links between the Mobile App and my IoT Devices.
5. As a User, I want to define Projects, so that I can group Topics and Dashboards, and organize the exchange of data between them.
6. As an IoT applications developer, I want to subscribe to the Topics, so that I can establish communication between my IoT Devices and the System.
7. As a User, I want to define Output Components that monitor the data incoming from the IoT Devices, so that I can visualize the current status of the IoT Device.
8. As a User, I want to define Input Components that send data to the IoT Devices, so that I can control their current state.

9. As a User, I want to integrate my IoT Devices with external mobile and web applications, so that I can utilize their functionalities in my IoT ecosystem.
10. As a User, I want to define Trigger Components that listen to specific Topics and respond with actions, so that I can create automated flows of information to external applications.
11. As a User, I want to get all the Messages, and Dashboard updates in real time so that I always have the current state of the System.
12. As a User, I want to share my Projects with other Users, so that I can facilitate collaboration.
13. As a User, I want to create an account, so that I can have restricted access to my resources.
14. As a User, I want to have an introduction to using the Mobile App, so that I know all the available features.
15. As an Admin, I want to block User accounts, so that harmful Users cannot use the System.
16. As an Admin, I want to manage privileges of other Users, so that there is always a trusted group of Admins.

Non-functional requirements

1. **Portability**

User stories:

- As a System Administrator, I want to run the Backend Server on different operating systems so it fits my current equipment.
- As a System Administrator, I want the Backend Server to be containerized so I can configure, and run it with minimal effort.
- As a User, I want to run the Mobile App on Android mobile phones so I can use it on a wide range of devices.

Acceptance criteria:

- The Backend Server should be portable across popular operating systems.
- The Backend Server should be containerized using a tool such as Docker.
- The Mobile App should be able to run on any Android mobile phone with Android version 26 and with average hardware specification.

2. **Performance**

User stories:

- As a User, I want the System to respond quickly to my actions so it is seamless to use.
- As a System Administrator, I want the Backend Server to close requests quickly, so they use as little system resources as possible.
- As a User, I want the System to send notifications and messages to external services continuously with little delay so I can be quickly notified of important events on my IoT devices.

Acceptance criteria:

- The Backend Server should respond to any request in less than 1.5 seconds in case of high load (1000 concurrent requests), and less than 1 second in case of normal load (500 concurrent requests).
- The Mobile App should fully process any user actions in less than 3 seconds in normal conditions (i.e. good Internet connection).
- The System should send messages and notification to external services in less than 60 seconds in case of e-mail, and in less than 15 seconds in case of all the other services.

3. **Reliability**

User stories:

- As a User, I want the System to respond to all my requests so I don't have to repeat my actions.
- As a User, I want the Messages to be always delivered so I don't miss any updates from the IoT Devices, and my actions aren't lost.
- As a User, I want my Messages to be delivered exactly once so the data is reliable.
- As a User, I want Messages to be delivered in correct order so I always have the most recent information.
- As a User, I want the Dashboard updates to be propagated in correct order so I always have the most recent Dashboard state.

Acceptance criteria:

- In the case of high load (1000 concurrent requests), the Backend Server should not respond to the maximum of 5% of requests.
- The data entered by the Users should be persisted indeterminately, unless it's deleted purposefully by user action.
- No Messages should be lost in a case of a normal load (500 concurrent messages).
- No Message should be unpurposefully sent twice in any conditions.
- The Messages should always appear to the User in the order they appeared on the Backend Server.
- All Dashboard updates should be propagated to the interested Users in the order they appeared on the Backend Server.

4. **Usability**

User stories:

- As a User, I want the Mobile App to inform me of any errors so that I have the right knowledge of the current state of the System.
- As a User, I want the Mobile App to alert me of any mistakes in my form so that I know what data needs to be corrected.
- As a User, I want the Mobile App to indicate that data is loading so there is no confusion about the state of the System.
- As a User, I want the Backend Server to not persist inconsistent data so that my user experience is stable.

Acceptance criteria:

- In case of errors of any kind (i.e. connectivity problems), there should be an error screen, or an error alert displayed in the Mobile App.

- In case of invalid data in any form field, there should be a clear warning displayed in the Mobile App.
- When the data is loading, the Mobile App should display the loading icon.
- The Backend Server should not persist data in the wrong format.
- The overall user satisfaction score should be at least 80%.

5. **Security**

User stories:

- As a User, I want my data to be protected from unauthorized access so that it stays private.
- As a User, I want my data to be protected from unauthorized modification so that it stays consistent with my needs.
- As a User, I want my password to be strong so no one can authenticate as me.
- As a User, I want my password to be hashed so it cannot be stolen.
- As an Admin, I want the System to not allow for any unauthorized actions, so that the state of the System stays consistent.

Acceptance criteria:

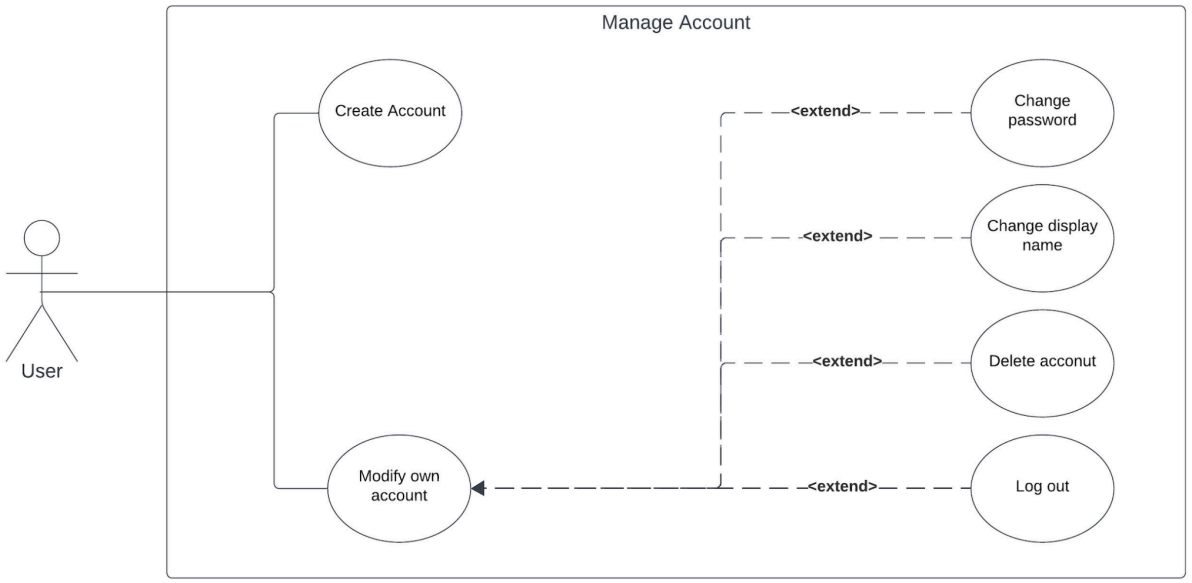
- Users' confidential data should be visible only to the selected Users.
- Users' resources should be modifiable only by the selected Users.
- All passwords must be at least 8 characters long.
- All passwords must be irreversibly hashed, and plain text (not hashed) passwords should not be stored.
- The System should authorize all requests, and reject any unauthorized requests.

Use case diagram

Use cases illustrate the exact examples of using the system to accomplish a specific task. They expand the user requirements by clarifying the scenarios and providing detailed descriptions of possible steps [19].

Manage Account

Fig. 1. Use case diagram showing accounting management.

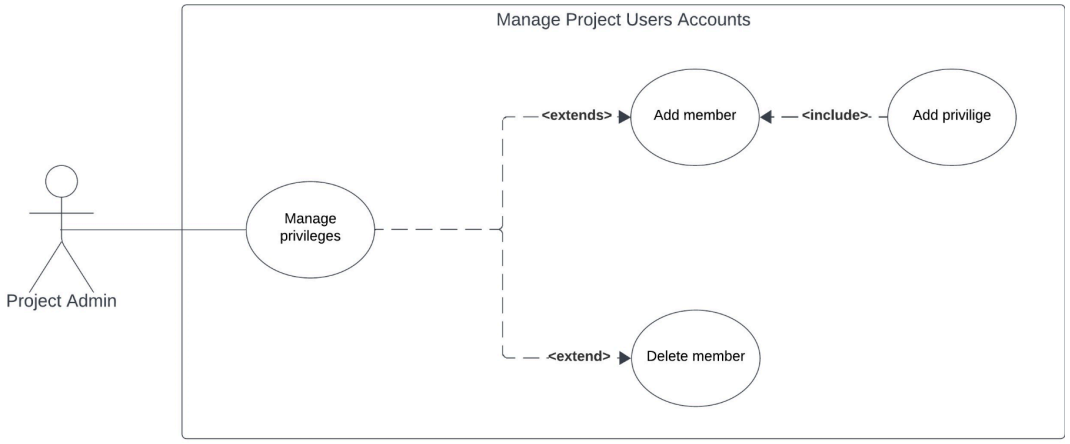


Use case description:

1. Refers to the User account.
2. Presents a set of management possibilities for the User.

Manage Project Users Accounts

Fig. 2. Use case diagram for management of users in a project.

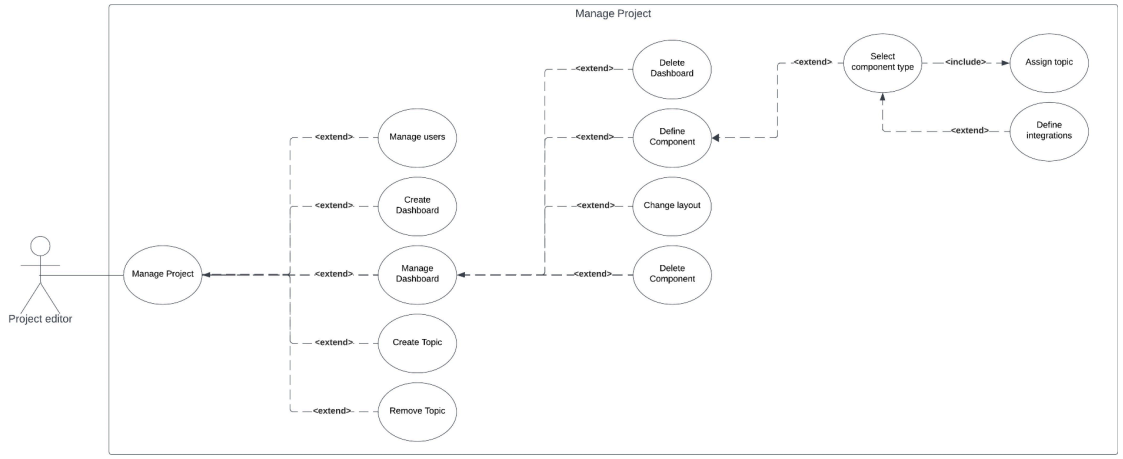


Use case description:

1. Refers to Project Admin.
2. Project owner is by default a Project Admin.
3. Is an extension of Manage Project Use Case Diagram in point *Change User Privileges.*

Manage Project

Fig. 3. Use case diagram for project management.

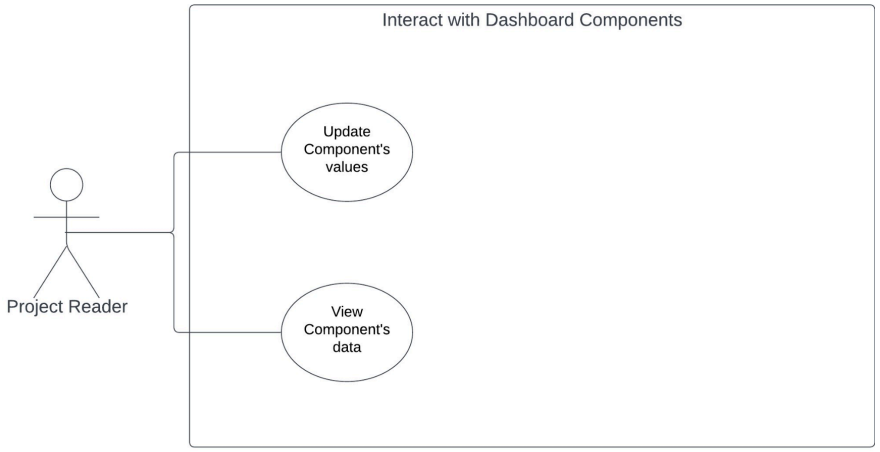


Use case description:

1. Refers to Project Editor including Dashboard and Component management.
2. Project Admin is by default Project Editor.

Interact with Dashboard Components

Fig. 4. Use case diagram for dashboard component interactions for a user with viewing privileges.

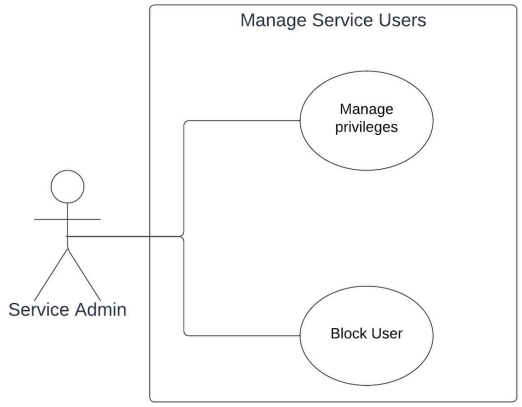


Use case description:

1. Regular interaction with the Dashboard.
2. Project Admins and Project Editors are by default Project Readers.

Manage Service Users

Fig. 5. Use case diagram for user management in the scope of the entire application.



Use case description:

1. Refers to service administration.

Use case specification

Create account

Brief description: User creates an account to use the System
Actors: Anonymous User

Preconditions:

1. Anonymous User opens the application for the first time or is logged out

Basic flow:

1. Anonymous User navigates into the “Register” screen
2. Anonymous User inputs name, username, password and password confirmation
3. Anonymous User confirms input data and registers
4. Anonymous User enters the code sent to given address email

Alternative flow:

4.1. Code was not sent

1. Anonymous User clicks “Resend” button
2. Use case resumes at point 4

Exceptional flow:

2.1. Username is already taken

1. Application prompts for another username
2. Use case resumes at point 2

2.2. Password doesn’t match password confirmation

1. Application prompts for password validation
2. Use case resumes at point 2

Action results:

1. Anonymous User has account created and can log in

Login

Brief description: User logs in to the System
Actors: Anonymous User

Preconditions:
1. Anonymous User opens the application for the first time or is logged out

Basic flow:
1. Anonymous User navigates into the “Login” screen
2. Anonymous User inputs his username and password
3. Anonymous User click “Login” button

Exceptional flow:
2.1. Username or password are incorrect
1. Login process fails
2. Use case resumes at point 2

Action results:
2. User is logged in

Create Project

Brief description: User creates a new Project
Actors: User

Preconditions:
1. User is properly logged in

Basic flow:
1. User navigates to “Projects” page
2. User clicks “Create project” button
3. User provides a name for the Project
4. User confirms Project creation

Alternative flow:
4.1. User doesn’t confirm
1. Project is not created
2. Use case resumes at point 2

Action results:
1. New project is created
2. New project is visible in “Projects” page

Create Dashboard

Brief description: Project editor creates Dashboard
Actors: Project editor

Preconditions:
1. User is properly logged in
2. User has an editor role in the Project

Basic flow:
1. Project Editor navigates to the Project and chooses “Dashboards” page
2. Project Editor click “Create dashboard” button
3. Project Editor provides a name for the Dashboard
4. Project Editor confirms Dashboard creation

Alternative flow:
4.1. Project Editor doesn’t want to create a Dashboard
1. Project Editor clicks “Cancel” button
2. Use case resumes at point 2

Action results:
1. New Dashboard is created in the Project

Create Topic

Brief description: User defines a Topic in the Project for communication with the IoT device
Actors: Project Editor

Preconditions:
1. User is properly logged in
2. User has an editor role in the Project

Basic flow:
1. Project Editor navigates to the Project and chooses “Topics” page
2. Project Editor click “Create topic” button
3. Project Editor provides a name for the Topic
4. Project Editor chooses the data type
5. Project Editor confirms Topic creation

Alternative flow:
4.1. Project Editor doesn’t want to create a Topic
1. Project Editor clicks “Cancel” button
2. Use case resumes at point 2

Exceptional flow:
5.1. Topic with such name in the chosen Project already exists

1. Topic is not created
2. Error message is displayed to the Project Editor
3. Use case resumes at point 2

Action results:
1. Topic is added to the Project

Define Input/Output Component

Brief description: User defines a new Input/Output Component in the chosen Dashboard
Actors: Project Editor

Preconditions:
1. User is properly logged in
2. User has an editor role in the Project

Basic flow:
1. Project Editor navigates to the Dashboard and clicks “Add component” button
2. Project Editor chooses component type from the list of Input/Output Components
3. Project Editor assigns Topic to the Component
4. Project Editor provides a name for the Component
5. Project Editor configures the Component (configuration form differs depending on type)
6. Project Editor confirms Input/Output Component creation

Alternative flow:

3.1. Project Editor creates new Topic
1. Project Editor creates a new Topic by clicking “Create topic” button
2. Flow is the same as in “Create Topic” use case from point 3
3. Use case resumes at point 4.

6.1. Project Editor doesn’t want to create a Component
1. Project Editor clicks “Cancel” button
2. Project is not created
3. If a new Topic was created in the process, it continues to exist
4. Use case is resumed at point 1

Action results:
1. New Input/Output Component is added into a Dashboard

Define Trigger Component

Brief description: User defines a new Trigger Component in the chosen Dashboard
Actors: Project Editor

Preconditions:
1. User is properly logged in
2. User has an editor role in the Project

Basic flow:

1. Project Editor navigates to the Dashboard and clicks “Add component” button
2. Project Editor chooses component type from the list of Trigger Components
3. Project Editor assigns Topic to the Component
4. Project Editor provides a name for the Component
5. Project Editor provides a pattern of the message/notification for the Component
6. Project Editor configures the Component (configuration form differs depending on type)
7. Project Editor confirms Trigger Component creation

Alternative flow:

3.1. Project Editor creates new Topic

1. User creates new Topic by clicking “Create topic” button
2. Flow is the same as in “Create Topic” use case from point 3
3. Use case resumes at point 4.

7.1. Project Editor doesn’t want to create a Component

1. Project Editor clicks “Cancel” button
2. Project is not created
3. If a new Topic was created in the process, it continues to exist
4. Use case is resumed at point 1

Action results:

2. New Trigger Component is added into a Dashboard

Add user to the Project

Brief description: User adds new user to the Project
Actors: Project Admin

Preconditions:

1. User is properly logged in
2. User is in the Project
3. User has an admin role in the Project

Basic flow:

1. Project Admin navigates to the Project and chooses “Topics” page
2. Project Admin goes to “Invite users” page
3. Project Admin inputs username of the user he wants to add
4. Project Admin chooses username from the list

Action results:

1. New user is invited to project
2. Invited user can see the received invitation

Respond to Invitation

Brief description: User responds to the Invitation
Actors: User

Preconditions:
1. User is properly logged in
2. User has received an Invitation to the Project

Basic flow:
1. User navigates to the “Invitation” page inside the “Account” page
2. User views existing Invitations
3. User clicks “Accept” button next to the selected Invitation

Alternative flow:
2.1. There are no Invitations
1. The empty page is displayed

3.1. User rejects the Invitation
1. User clicks the “Reject” button next to the selected Invitation
2. Use case resumes at 2

Action results:
1. Invitation is deleted (in case of both basic and alternative flow)
2. User gains an access to the Project

Revoke dashboard access

Brief description: User revokes access to the Project for a different User
Actors: Project Admin

Preconditions:
1. User is properly logged in
2. User is in the Project
3. User has an admin role in the Project

Basic flow:
1. Project Admin navigates to the Project and chooses the “Group” page
2. Project Admin goes to “Revoke access” page
3. Project Admin chooses “remove” next to the username to be revoked access
4. Project Admin is prompted for confirmation of access removal
5. Project Admin confirms his action

Alternative flow:
4.1. Project Admin doesn’t want to remove chosen member
1. Project Admin doesn’t confirm member removal
2. Use case resumes at point 1

Exceptional flow:
5.1. Chosen member is the Project Owner
1. Project Admin is alerted about the operation failure
2. Use case resumes at point 1

Action results:
1. Member has no longer granted access to the Project
2. Member is not longer visible in a group view

Manage user privileges

Brief description: User assigns user privileges for the Project
Actors: Project Admin

Preconditions:
1. User is properly logged in
2. User is in the Project
3. User has an admin role in the Project
4. Target User has any role in the Project

Basic flow:
1. Project Admin navigates to the Project and chooses the “Group” page
2. Project Admin goes to “Edit roles” page
3. Project Admin chooses the target member from the list of members
4. Project Admin chooses a new role for the target member
5. Project Admin confirms his action

Alternative flow:
4.1 Project Admin doesn’t want to edit chosen member
1. Project Admin doesn’t confirm member editing
2. Use case resumes at point 1

Exceptional flow:
5.1. Chosen member is the Project owner
1. Project Admin is alerted about the operation failure
2. Use case resumes at point 1

Action results:
1. Target member’s role is changed

Remove Topic

Brief description: User removes Topic from the Project
Actors: Project Editor

Preconditions:

1. User is properly logged in
2. User has an editor role in the Project

Basic flow:

1. Project Editor navigates to the Project and chooses the “Topic” page
2. Project Editor clicks “Delete” button next to the selected Topic
3. Project Editor clicks “Confirm” button

Alternative flow:

3.1. Project Editor doesn’t want to remove the Topic

1. Project Editor clicks “Cancel” button
2. Use case resumes at point 1

Exceptional flow:

3.1. Selected Topic is used

1. The error message is displayed to the Project Editor
2. Use case resumes at point 1

Action results:

1. Topic is removed from the Project

Remove Dashboard

Brief description: User removes Dashboard from the Project
Actors: Project Editor
Preconditions:

1. User is properly logged in
2. User has an editor role in the Project

Basic flow:

1. Project Editor navigates to the Project and chooses the “Dashboards” page
2. Project Editor enters the selected Dashboard
3. Project Editor clicks “Delete” option from the option list
4. Project Editor clicks “Confirm” button

Alternative flow:

4.1. Project Editor doesn’t want to remove the Dashboard

1. Project Editor clicks “Cancel” button
2. Use case resumes at point 1

Action results:

1. Dashboard is removed from the Project
2. All Components within the Dashboard are removed

Edit layout

Brief description: User edits Dashboard's layout
Actors: Project Editor

Preconditions:
1. User is properly logged in
2. User has an editor role in the Project

Basic flow:
1. Project Editor navigates to the Project and chooses the "Dashboards" page
2. Project Editor enters the selected Dashboard
3. Project Editor clicks the "Edit" option from the option list
4. Project Editor moves the Components to desired positions
5. Project Editor click the "Return to the normal mode" button

Action results:
1. Components within the Dashboard are rearranged

Remove Component

Brief description: User removes a Component
Actors: Project Editor

Preconditions:
1. User is properly logged in
2. User has an editor role in the Project

Basic flow:
1. Project Editor navigates to the Project and chooses the "Dashboards" page
2. Project Editor enters the selected Dashboard
3. Project Editor clicks the "Edit" option from the option list
4. Project Editor click the "Delete" button on the selected Component

Action results:
1. Component is deleted

Remove Project

Brief description: User removes their Project
Actors: Project Owner

Preconditions:
1. User is properly logged in
2. User is an owner of the Project

Basic flow:

1. Project Owner navigates to the selected Project
2. Project Owner clicks “Delete” option from the option list
3. Project Owner clicks “Confirm” button

Alternative flow:

3.1. Project Owner doesn’t want to remove the Project

1. Project Owner clicks “Cancel” button
2. Use case resumes at point 1

Action results:

1. Project is removed
2. All Components, Dashboards, and Topics within the Project are removed

---

Change password

Brief description: User changes their password
Actors: User

Preconditions:

1. User is properly logged in

Basic flow:

1. User navigates to the “Account” page
2. User clicks the ”Change password” option from the menu
3. User inputs a new password
4. User repeats the new password
5. User clicks the “Confirm” button

Alternative flow:

5.1. User doesn’t want to change the password

1. User clicks the “Cancel” button
2. Use case resumes at point 2

Exceptional flow:

3.1. Password does not match password policy

1. User is prompted about the password policy
2. Use case resumes at point 3

4.1. Repeated password does not match the new password

1. User is prompted about the mistake
2. Use case resumes at point 4

Action results:

1. User’s password is changed

Change display name

Brief description: User changes their display name
Actors: User

Preconditions:
1. User is properly logged in

Basic flow:
1. User navigates to the “Account” page
2. User clicks the ”Change display name” option from the menu
3. User inputs the new name
4. User clicks the “Confirm” button

Alternative flow:
4.1. User doesn’t want to change the display name
1. User clicks the “Cancel” button
2. Use case resumes at point 2

Action results:
1. User’s display name is changed

Block user account

Brief description: Admin blocks User account
Actors: Admin
Preconditions:
1. Admin is properly logged in

Basic flow:
1. Admin navigates to the “Admin” panel and chooses the “Block users” option
2. Admin clicks “Block” button next to the selected User

Action results:
1. User account is blocked
2. “Block” button toggles to “Unblock” button

Unblock user account

Brief description: Admin unblocks User account
Actors: Admin

Preconditions:
1. Admin is properly logged in

Basic flow:

1. Admin navigates to the “Admin” panel and chooses the “Block users” option
2. Admin clicks “Unblock” button next to the selected User

Action results:

1. User account is unblocked
2. “Unblock” button toggles to “Block” button

Add admin

Brief description: Admin grants admin privileges to another User
Actors: Admin

Preconditions:

1. Admin is properly logged in

Basic flow:

1. Admin navigates to the “Admin” panel and chooses the “Add admin” option
2. Admin clicks “Add” button next to the selected User

Action results:

1. Target User is granted Admin role

Update Component’s values

Brief description: User updates Component’s values
Actors: Project Reader

Preconditions:

1. User is properly logged in
2. User has a reader role in the Project

Basic flow:

1. Project Reader navigates to the selected Dashboard
2. Project Reader updates the selected Component’s values by interacting with the Component (interaction mechanism differs depending on the type of the Component)

Alternative flow:

2.1. There are no Components

1. Empty Dashboard is displayed

Action results:

1. Component’s values are updated

View Component's data

Brief description: User views Component's data
Actors: Project Reader

Preconditions:
1. User is properly logged in
2. User has a reader role in the Project

Basic flow:
1. Project Reader navigates to the selected Dashboard
2. Specific Component's data is visible to the Project Reader in a way specified by the Component's type (i.e. in a graph form for graph Components)
3. Project Reader clicks the "Information" button on the Component to view its detailed data (optional step)

Alternative flow:
2.1. There are no Components
2. Empty Dashboard is displayed

Action results:
None

7. Software product design

Architecture

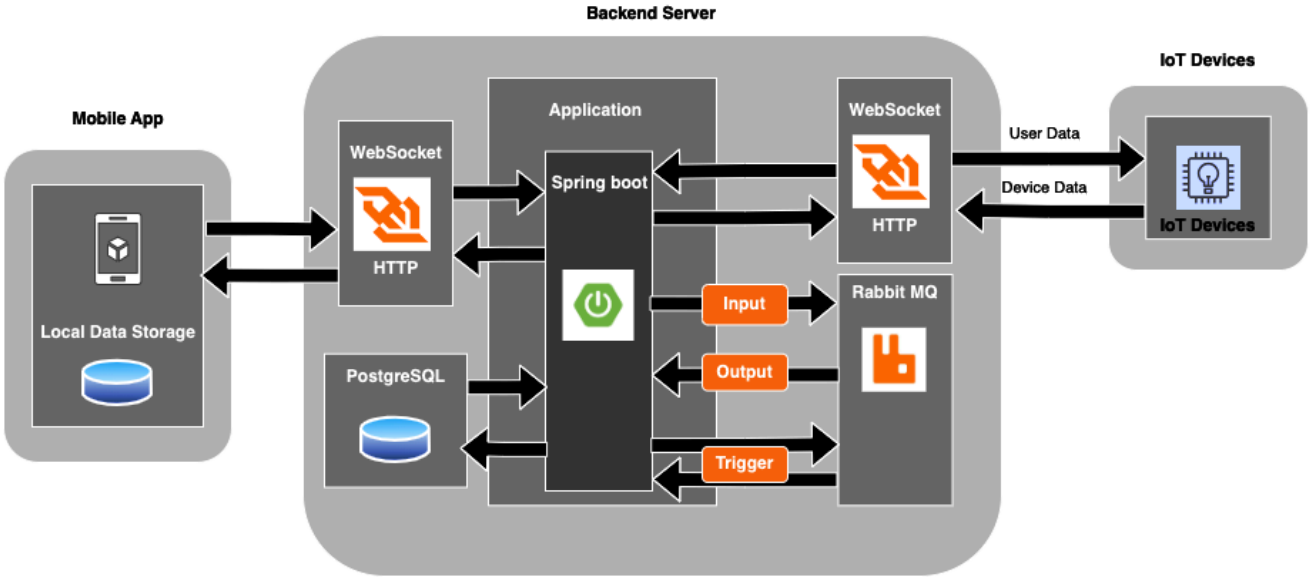
The architecture was designed to provide the project with the following qualities [20]:

- Clarity. Architecture is an abstraction of the system that the stakeholders can use to better understand the project.
- Error preventability. Architecture allows the developers to analyze systems in early stages of design in terms of development time, or feasibility.
- Direction.The architecture defines constraints on implementation. It gives the developers a first idea of how the implementation process should proceed.

In the following sections, the general components present in the System, and the links of communication between them, are defined. Then, there are detailed descriptions of architectures of particular parts of the System.

Components

Fig. 6. The architecture diagram of the project showing system components and direction of data transmission.



Above diagram roughly illustrates the main components of the System, and the communication links between them.

The central part of the System is the Backend Server. It consists of the Spring application that we created. It's important to note that there is only one instance of the Spring application. There are also additional services connected to the Spring application (each of them also has one instance running), specifically:

- PostgreSQL database - responsible for data persistence, it is connected only to the Spring application.
- RabbitMQ server - responsible for internal message distribution. Provides the System with reliable, and fast message delivery. It is also connected only to the Spring application.

The Mobile App is on the left side of the diagram. It is the main interface via which the Users are going to interact with the System. Many Mobile Apps are connected to the same Backend Server, and one instance of the application is equivalent to one User.

The IoT devices are represented on the right side of the diagram. There will also be a lot of instances of the IoT devices, each connected to the same Backend Server. The idea is, many Users are able to interact with the same IoT devices via their Mobile App instances, and each User may interact with multiple devices at the same time.

Communication

As there are a few components in the System, it's necessary to define the communication links between particular parts, the nature of transported data, and the internal flow of information [20].

The communication between the Mobile App and the Backend Server is conducted with the use of two protocols: HTTP and WebSocket protocol.

HTTP is used in a typical request-response manner. The Mobile App always acts as a client, while the Backend Server as a server [21]. The Mobile App initiates the interactions, requesting resource retrieval, or modification. The Backend Server sends back appropriate resources and status codes.

We utilize the WebSocket protocol to facilitate continuous communication through an established connection. This protocol is essential for providing the Mobile App with real-time updates about database changes or with Messages from IoT devices [22]. The Mobile App initiates the two-way communication, and upon establishing the connection, it sends an initial message to request the stream of data from the Backend Server. This communication model is inspired by the "client-requests-a-server-stream" approach available in RSocket protocol [23].

Backend

The Backend Server follows three-tier Controller-Service-Repository architecture. Repository layer provides the service layer with functions to retrieve and modify persistent data. The service layer is responsible for all the logic. The controller layer provides the REST API that can be accessed and utilized by the external applications (Mobile Apps and IoT Devices) [24].

The REST API is all about resource access and management. The available endpoints allow for retrieving and modifying persistent data in the Backend Server using the HTTP protocol [25]. Most endpoints are intended for the Mobile App instances, while a few are reserved for the IoT devices. The URIs for the endpoints have a specific format:

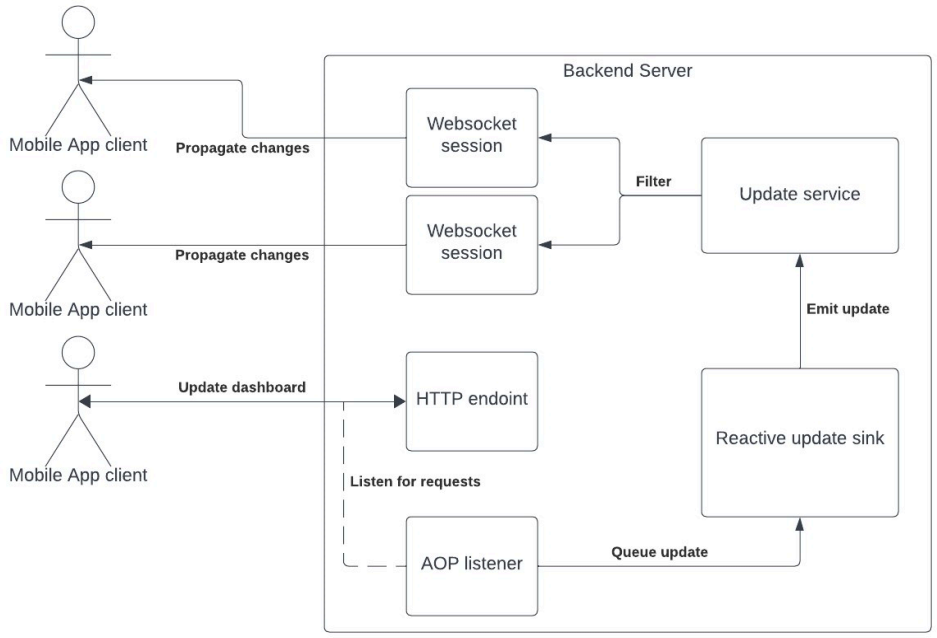
*/{user_role}/{area_of_concern}/***

User_role defines what is the minimal role to use this endpoint. Area_of_concern specifies what is the main subject of the accessed resource. This path design, together with the server-client model, introduces the separation of concerns to the System. The endpoints available for the IoT devices always have *user_role* = anon, and a term "device" at the end of the path but before the path variables.

Except for the REST API, there are WebSocket interfaces available for both the Mobile Apps and IoT devices. These are used for propagating persistent updates, and Messages in real time to the interested parties.

To propagate changes in the persistent data, we used aspect oriented programming. It further promotes the separation of concerns [26]. In Spring, this technique is implemented using proxies [27]. When there is a request that modifies an observed resource, it is intercepted, and logic is added to propagate changes to all interested parties. The changes are transported to the WebSocket session via reactive stream. This is so called publisher-subscriber patterns, and it allows the System to consume resources only when there are changes to propagate [28]. The following diagram roughly illustrates how the change propagation looks like.

Fig. 7. Diagram showing change propagation and synchronization between multiple systems.



There is also the integration manager that handles all events related to the integrations with external services. It is implemented using a singleton design pattern, as there should be just one manager for all defined integrations [29]. These integrations are available through Trigger Components. They work in the following manner:

1. The Message appears on the Topic selected by the User defining the Trigger Component.
2. The notification is formatted using the pattern given in Trigger Component configuration, the data from the Message, and some additional data.
3. The notification is sent to an external service determined by the type of Trigger Component.

Mobile App

The mobile application architecture plays a crucial role in the systems performance, code structure and maintenance. A clearly defined architecture makes code easier to develop, less prone to errors, easier to test and maintain.

Mobile applications usually consist of layers that group certain processes together. In particular, a typical Android application consists of three layers – the presentation, business and data layer.
The presentation layer handles the user interface (UI) and user experience (UX). It takes care of the visual part of the software, as well as all the interactions between user and the system.
The business layer is used for processing data, managing state, handling errors, validating data and adding functionality to the application. It contains the logic concerning data exchange between the systems.
The data layer is responsible for gathering and updating data. It concerns the persistence of data, as well as network communication. It handles both local and remote data sources, and transfers the information to lower layers [30].

The above three layers constitute the clean architecture, in which the crucial concept is avoiding the codependency of layers. In particular, the business logic should be separated from all other parts of the system, therefore the business layer should be separated from the other layers, and should rather communicate using interfaces.
This architecture works well with the Model-View-ViewModel (MVVM) design pattern widely used in Android app development. Such a structure of components separates the user interface from the application logic.
The models are classes that hold information. The view is the set of visual elements that the user interacts with. The view model is typically a class that manages and changes information from the model to be represented in the view. It is the element that connects the model with the view, and handles the state of the application [31].

The advantages of the MVVM pattern is the separation of concerns achieved by UI code separate from the app logic, simplified maintenance of code, as it allows to change a part of code without influencing another, and lastly increased usability as it promotes division of problems into smaller parts.

Both the clean architecture as well as MVVM pattern utilize modularization. From the Android documentation, “Modularization is a practice of organizing a codebase into loosely coupled and self contained parts” [32]. It enforces code reusability and allows for strict visibility control of classes.

Commonly used in Android development is the dependency injection (DI) pattern, in which “objects are associated with the instances of classes they are dependent on” [33]. It works on the basis of passing class references as parameters in another dependent class. It stands in place of creating your own class each time it is required, or retrieving it using the context of the application [34]. A significant advantage of DI pattern is loose coupling which allows for adding and testing new objects independently from others [33]. Therefore, it simplifies maintenance of code and increases its reusability.
In our project, dependency injection is handled by the Koin library which allows us to create modules with classes that are automatically transferred to objects that require them as parameters.

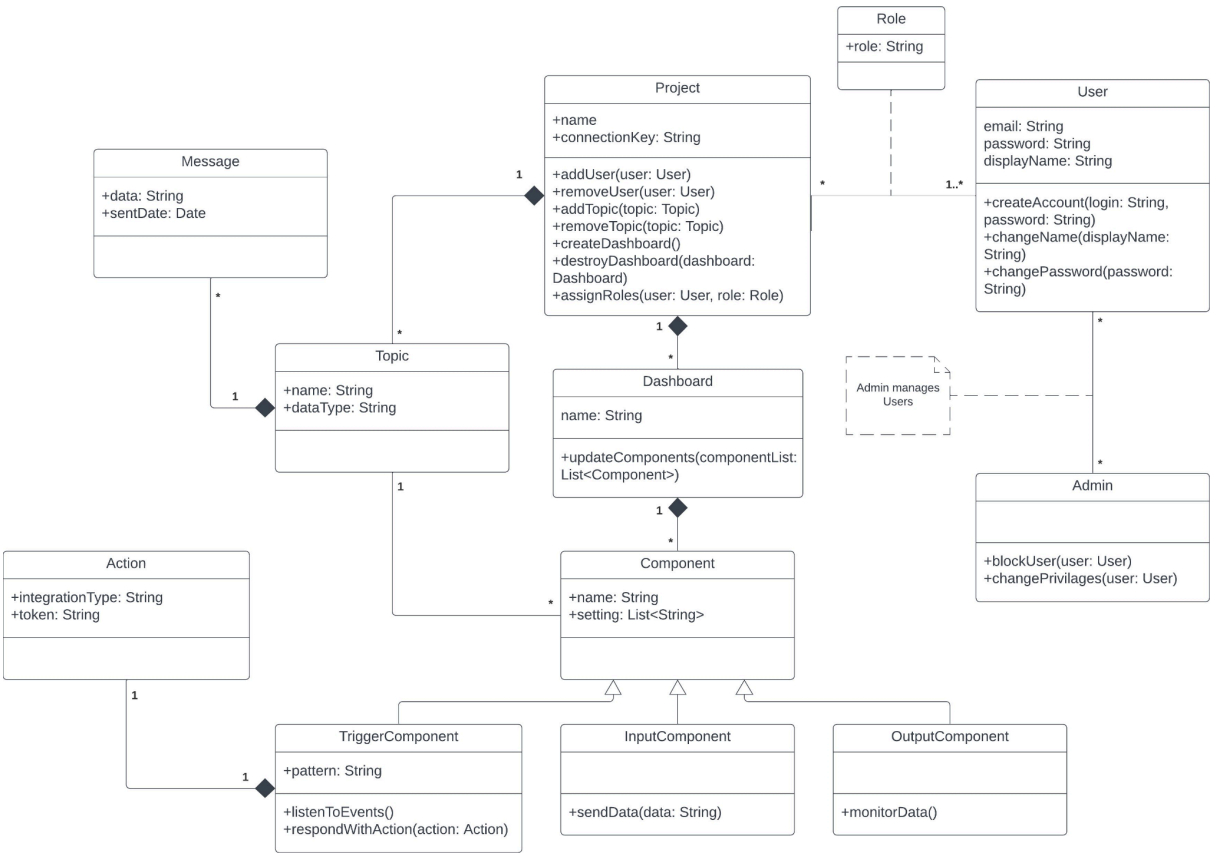
The newest trend in user interface development is using the declarative programming paradigm with Jetpack Compose. It focuses on dividing the UI into functions that render specific elements on the screen. When there is an update in the state of UI, only the elements using this particular state are updated by calling the function creating them again and therefore regenerating the part of view from scratch. This process is called recomposition. This paradigm is the opposition of the previous imperative programming paradigm, in which the visual elements on the screen could be represented in a tree-like structure. On the change of UI state, the view was updated by walking the tree and updating the particular elements which was heavy for screens with many of components, as well as error prone, as it allowed for creating invalid states. Jetpack compose greatly simplifies the development process and avoids the complexity connected to manually updating views [35].

The above design principles have been implemented into this project, allowing for a clear, scalable and easy to maintain architecture of the system. The detailed description of the usage of those rules is discussed in the Implementation part of this document.

Class diagram

Class diagram in UML is a structure diagram showing the structure of developed software. It represents software elements and the relations between them, as well as the data used by classes consisting of attributes and methods. The diagram can resemble both overall project structure, as well as detailed sections [36].

Fig. 8. The class diagram of the system.



The diagram above shows the class diagram of our application. Project is the core class which organizes Dashboards, Topics, and Users into one common space. It is managed by Project Admins and can be influenced by Project Editors.

The Dashboard class is used for grouping Components. It provides a method of organizing the component list so that it can be personalized for each use case. The user can create, delete and update the position of the Component in the list. A project can have many dashboards, each with a separate set of components.

The Component class provides the ability for communication between the User and the IoT device. It is the interface with which the user interacts to send and receive components. A Component must be assigned to a topic in order to be functional. There are 3 types of Components:

- Input Component - sends messages from the Mobile App to the IoT devices
- Output Component - receives data from the IoT devices, and presents it to the User on the Mobile App
- Trigger Component - enables integrations with external services such as Discord, Telegram, Notifications.

The Topic class is a communication channel for sending and receiving information. A single topic may hold many messages. The same topic may be used for many components, however, a single component can only have a single topic. The user may create many topics in a single project.
Lastly, the User class resembles the person who uses our application. Users can create and, with appropriate privileges, manage projects, invite others and use dashboards. The users have different roles in the application and project scope. A project must have at least one User, and may have many users in total. Admin is a special User with different privileges, and can take different actions from the regular User. There are two types of roles in the system:
Roles within **application scope**:

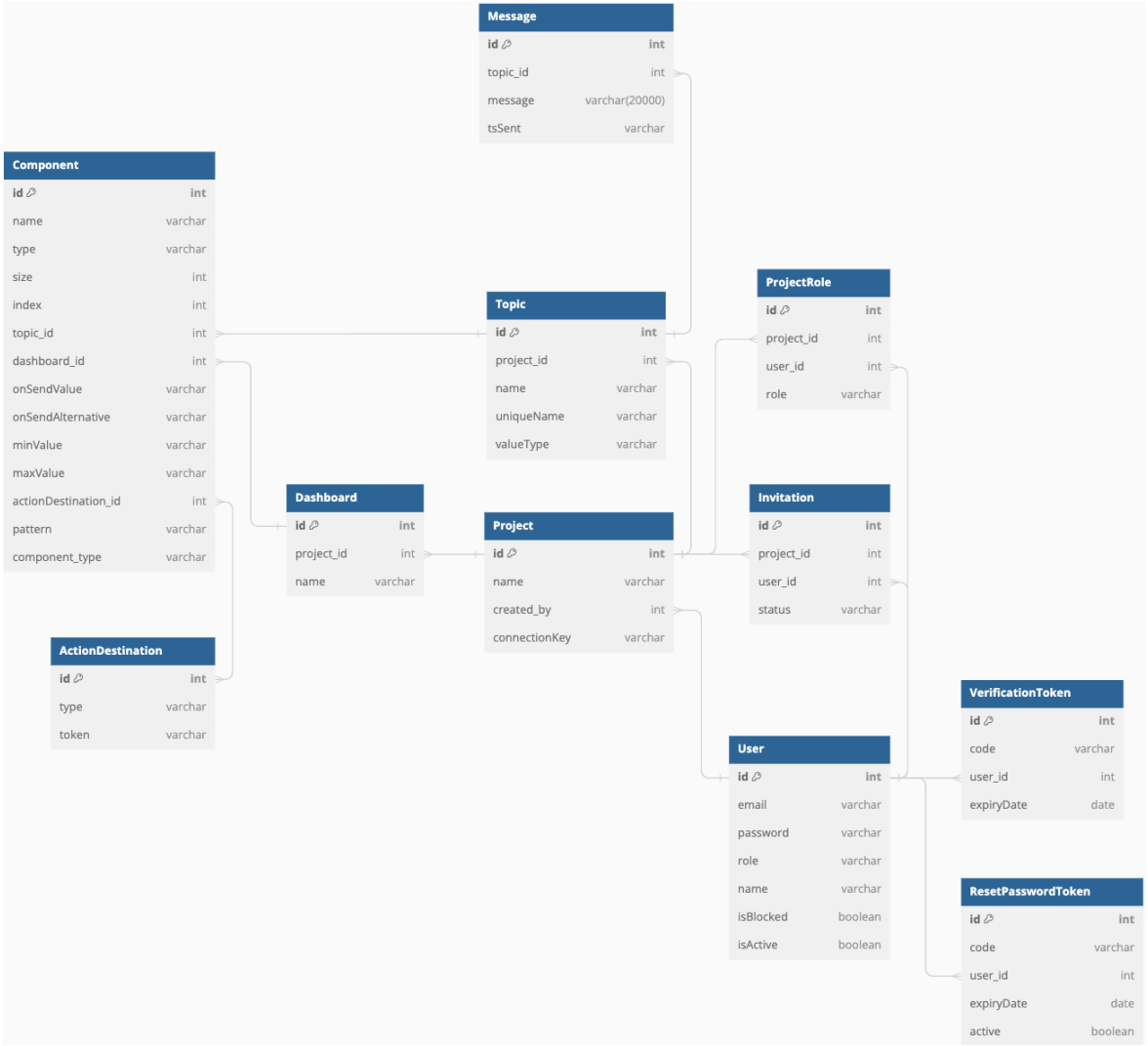
- **User** - normal user, account can be created by registration.
- **Admin** - role added by other existing admin. Can block, unblock and manage users.

Roles within **project scope**:

- **Viewer** - can access, interact with Dashboards and Components.
- **Editor** - can modify Projects, modify and add Dashboards, Components and Topics.
- **Admin** - can modify Projects, manage Group members, modify and add Dashboards, Components and Topics.
- **Owner** - has all possible privileges in the project. Set automatically when creating the project.

Database design

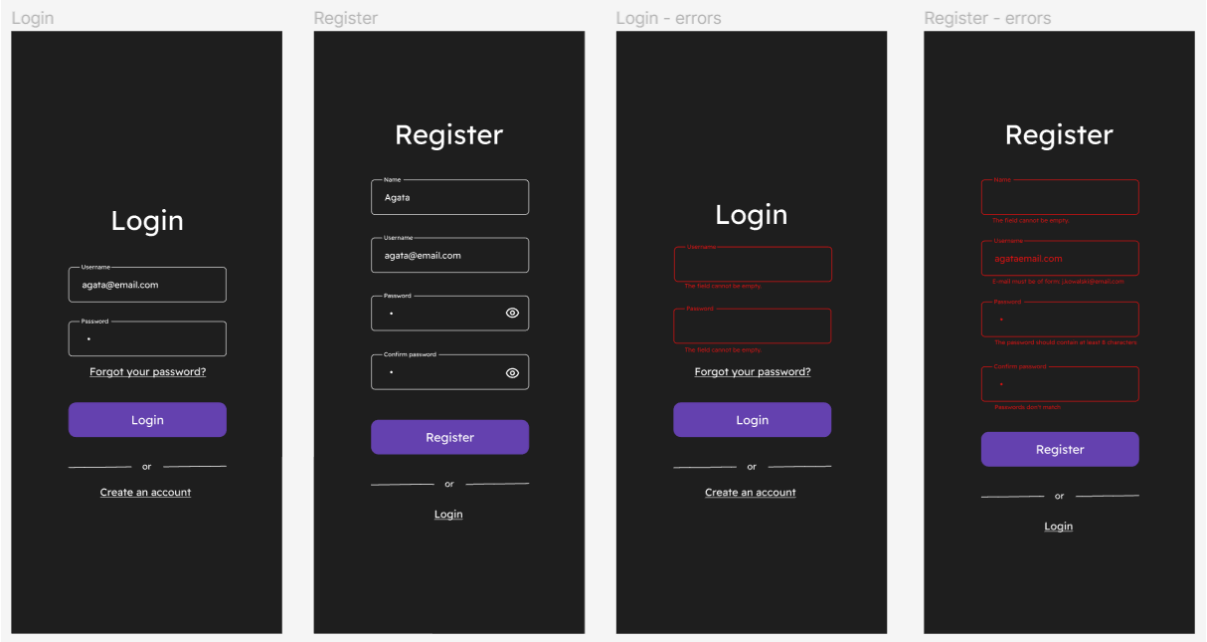
Fig. 9. The database diagram of the system.



Mockups

Login and registration

Fig. 10. The login and registration mockups.

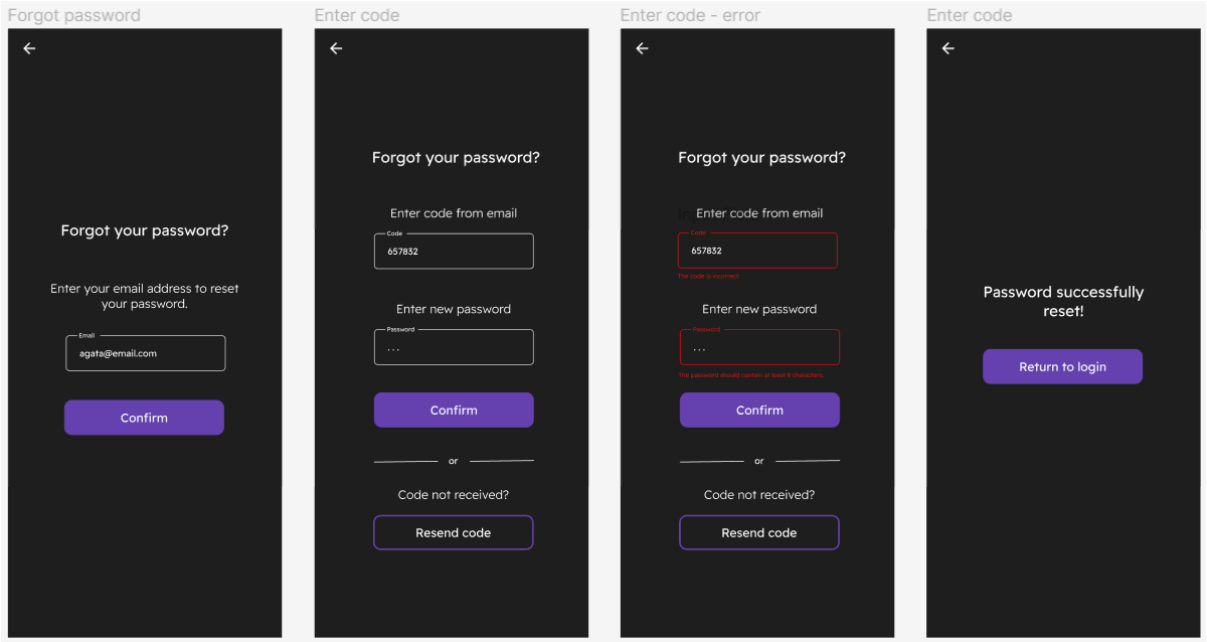


The login and registration pages allow the user to authorize within the application and personalize their experience. The process of login and registration should be simple and fast to complete so as to not distract the user from accessing the core functionality of the application. The screens have been, therefore, designed to be minimal and simplistic.
The focus of the screens are the input fields. They are in the center of the screen so that the user knows immediately what data to input. Another characteristic is the main button with contrasting color that brings attention to the main functionality of the given view.

On the login page, upon clicking the *Login* button, the user is authorized and then automatically navigated to the main screen of the application. If there was an error with the data in input fields, the appropriate ones turn red and display an error message.
This screen additionally has two clickable texts that navigate to other screens - the *Forgot your password* text which leads to retrieving the password page, and *Create an account* which navigates to the register screen.

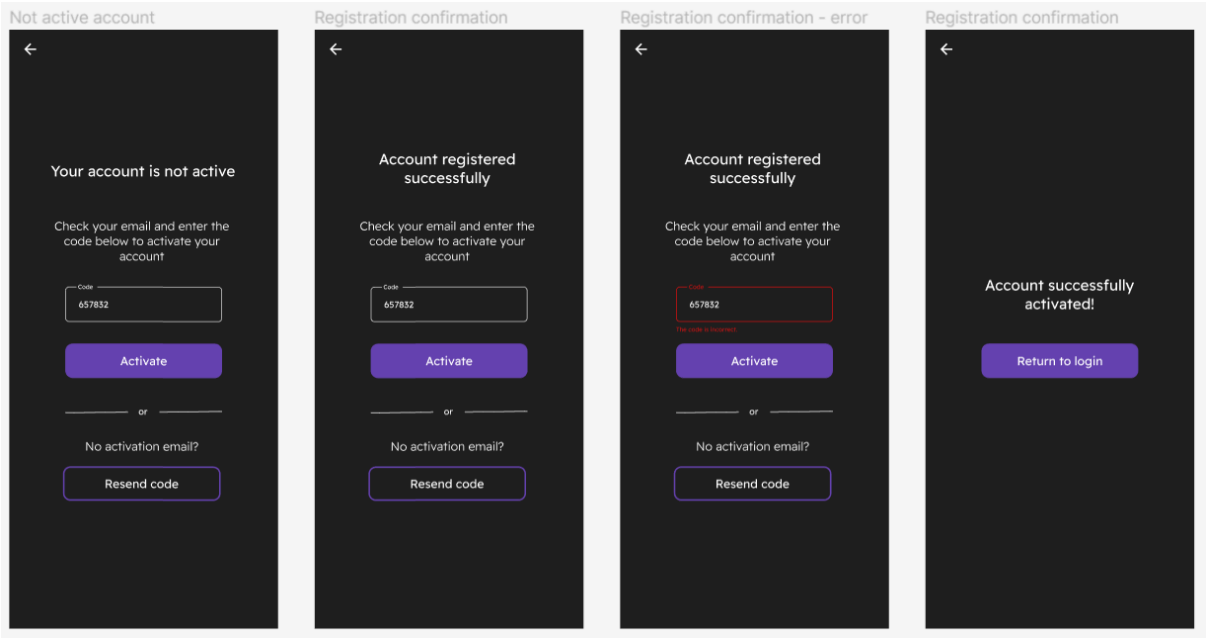
The register page is structured very similarly to the login page, however, it contains additional input field types for password. On default, the text is hidden, however when the user clicks on the eye on the right side of the screen, the input field shows its text value. Upon clicking *Register* the user is navigated to the Activate account screen, where he has to enter a code from email. The user can also return to the login page using the link text.

Fig. 11. The forgot password screen mockups.



The forgot password screen can be opened from the login page, and allows the user to quickly return to that screen through the arrow in the upper left corner. Upon entering, the screen asks for the user's email, and when he presses *Confirm*, the screen changes to the second from the fig. 11., where the user is asked to enter a code from email and a new password. If the user did not receive a code, he may resend the code by pressing the button on the bottom of the screen. The button has an invisible background and is separated using a divider because it serves as an additional functionality and should not take the user's focus of completing the main task. After pressing the confirmation button a second time, in case of a successful event the last page from the figure above is shown, and the user may return to login.

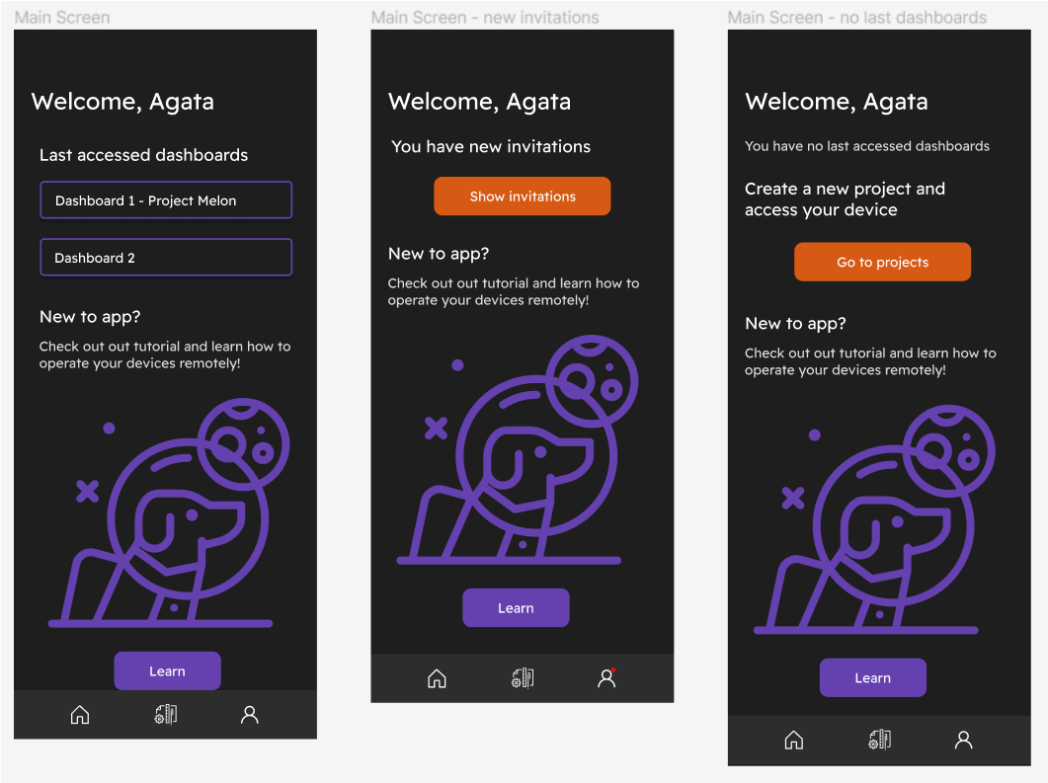
Fig. 12. The account activation screen mockups.



The activate account screen has a similar layout to *Forgot password* page, as it also serves as an additional, supplementary functionality. The user is asked to enter a code from email, and may resend code in a similar manner as in the previously discussed screen. Upon entering an incorrect code, the user is informed of the error near the input field. After successful activation, the last screen from the figure above is displayed, and the user returns to login.

Main screen

Fig. 13. The main screen mockups.



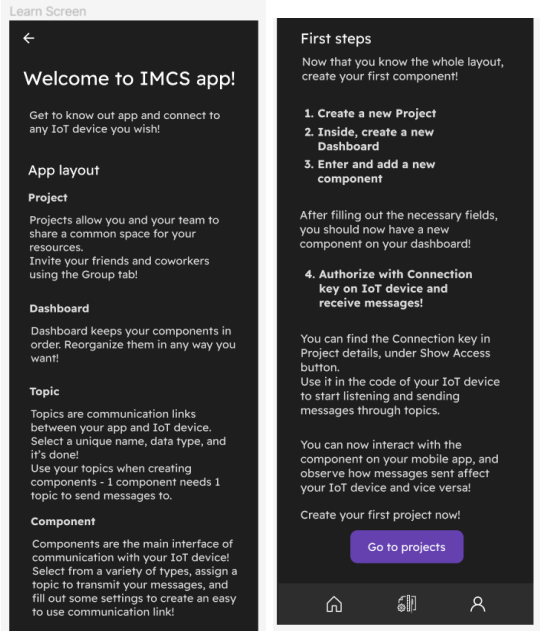
Main screen is the focal point of the application. It does not contain any core functionalities, however, it is the screen that welcomes the user, and should encourage him to further use the application, as well as simplify the use of the core functionalities for a long-term user. This screen is also good for notifying users of any new information or events, such as invitations.
When first entering the page, the focus is on navigating the user to the core functionality through the *Go to projects* button, and teaching the user about the application through the tutorial which can be accessed by the *Learn* button.
If the user has been using the app before, on the top of the screen there are shown the last accessed dashboards, upon clicking which the user is navigated to the appropriate dashboard. This greatly accelerates gaining access to the core functionalities, which is very useful for users who often come back to the application.

Additionally, if the user has any pending invitations, the button navigating to them will be visible at the top of the screen.

Lastly, the large title on the top of the page containing the name of the user is used to make users feel more welcome in the application, as well as it enhances the personalized experience throughout the application.

Main screen

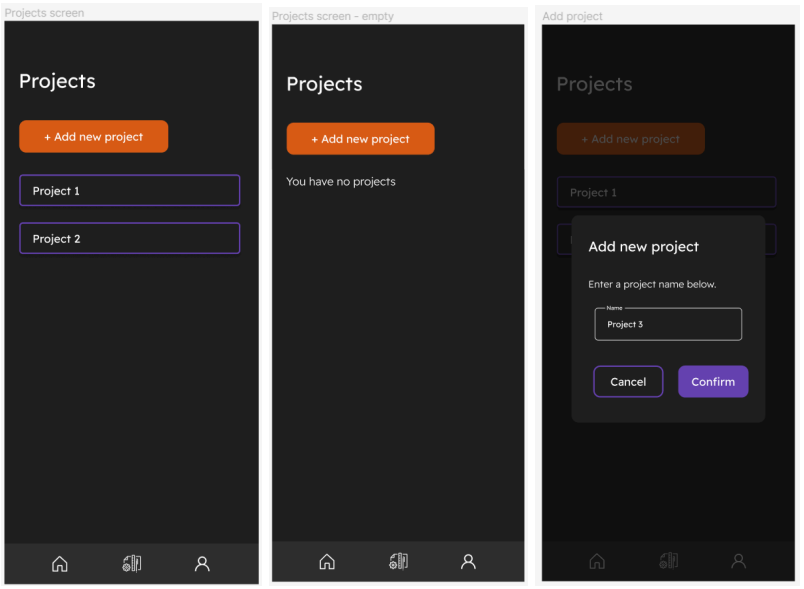
Fig. 14. The learn screen mockups.



The learn screen serves as an introductory tutorial and informative space for the new user. It describes the core parts of the application, and provides a set of initial steps to take when beginning working with the application. It navigates to the project screen using the *Go to project* button.

Projects screen

Fig. 15. The projects screen mockups.



The projects screen is a simple list of all of the projects the user has access to. By clicking the orange button, a dialog pops up that allows the user to create a new project by entering a name and pressing confirm. The cancel button closes the dialog without creating the project. After the user clicks on any of the items in the list, he is directed to the according project details screen.

Project details screen

Fig. 16. The project details screen mockups for the admin user.

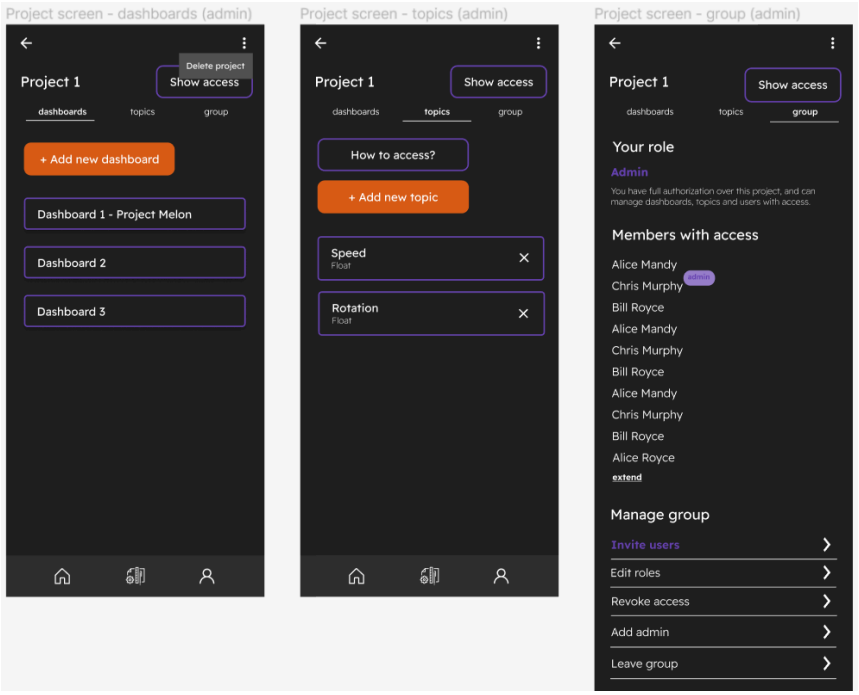


Fig. 17. The project details screen mockups for the viewer user.

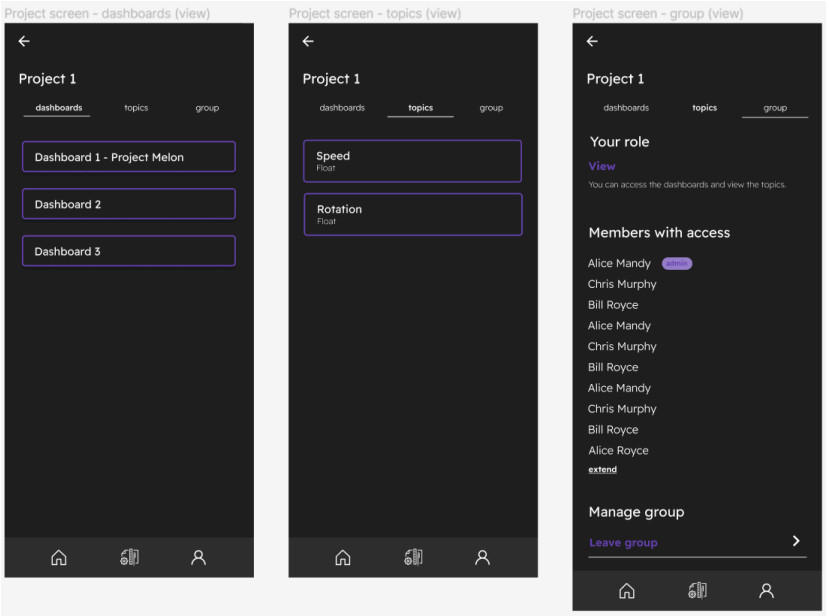
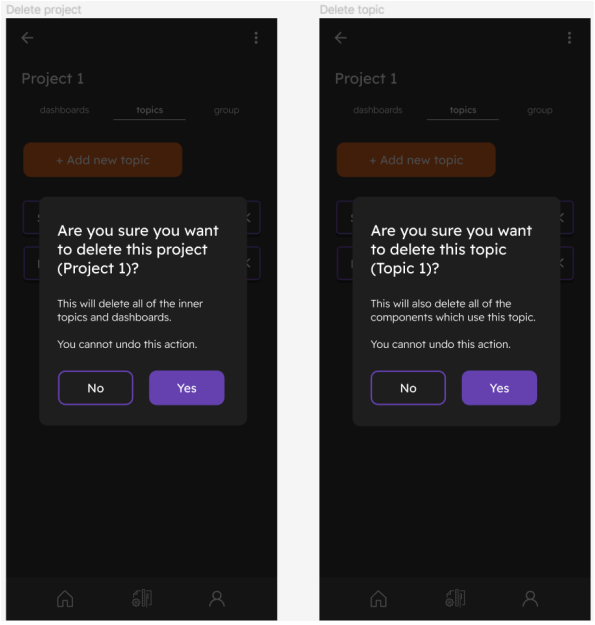


Fig. 18. The mockups of delete dialog for dashboard and topic.



The project details page is a complex screen which displays a large amount of important information. It is organized into tabs, by pressing which appropriate content is activated. The dashboard tab shows the list of created dashboards in the project, the topics tab shows a list of topics, and the group tab shows the group of project members with some menu items with special functionality below.
The screen varies in content for users of different roles: viewer, editor and admin. The viewer user is the most limited, therefore, many functionalities such as adding and deleting are hidden.
Upon pressing any item in the dashboard list, the user is navigated into the Dashboard screen. Pressing on the X mark next to topics opens a delete dashboard for the chosen topic.
The group menu rows all possess different purposes, with most navigating to the Search Screen, with the exception of the Leave *group* item which opens a pop up asking the user to confirm. The view also shows a top bar menu if the user is an admin.

Dashboard screen

The dashboard screen contains the core functionality of the application - components which serve as interface between the application and an IoT device. The view was made to be as simple as possible to avoid overcomplication due to the component nature. The screen is made of a list of components, each of different characteristics. The view comes in several modes - for viewers, editor users, and with an edit mode. The edit mode is only accessible to editor and admin users, similarly with the add component button. Every user can interact with the components by pressing them - pressing input components results in sending data. In edit mode, the user can reorder components by long pressing and dragging them around the screen. The app automatically calculates the new position of the component and places it using an animation.

Pressing on the info button in the top right corner of each component opens the info dialog. This icon turns into an X in edit mode. Upon pressing it, the delete component dialog is shown.

Fig. 19. The dashboard screen mockups for the different users, and the component information dialog.

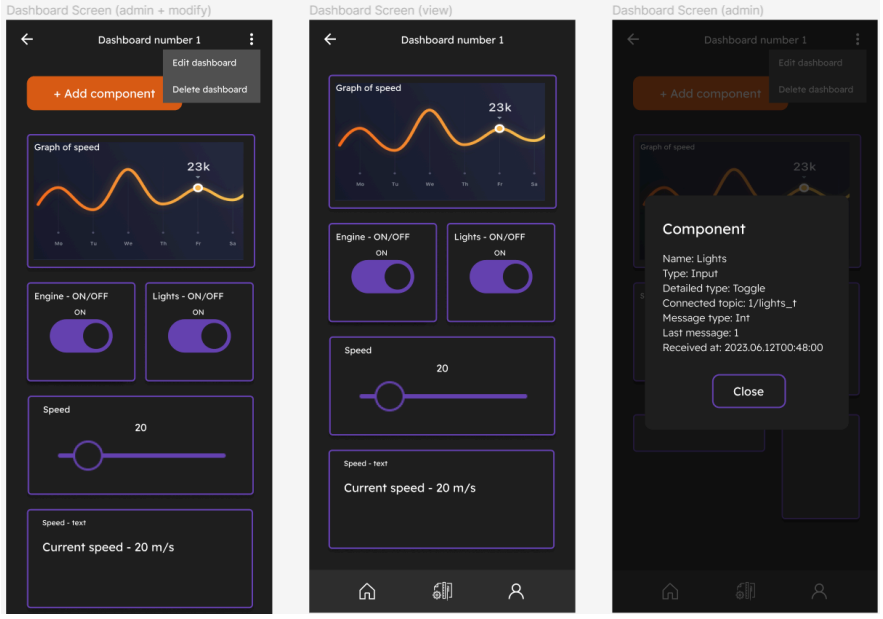
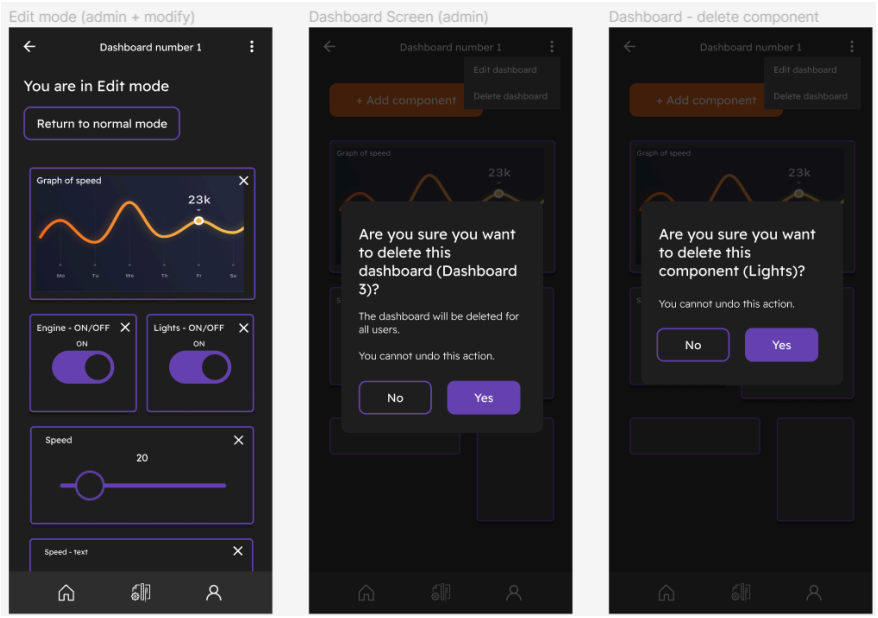
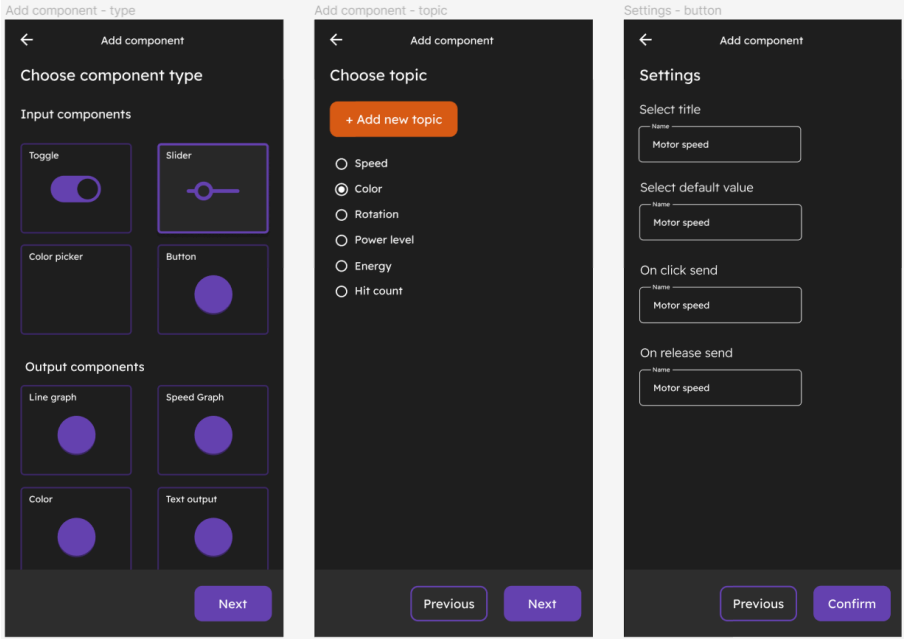


Fig. 20. The dashboard edit mode and delete dialogs.



Add component screen

Fig. 21. The add component screen mockups.

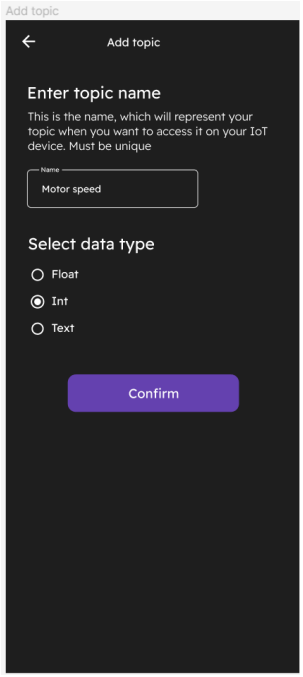


This screen has its own bottom navigation which is used to progress through steps of the process. Initially, the user must select a component from a list of available ones. If not pressed anything, the user is informed with a toast message that he must select at least one item. By pressing next, the *Choose topic* screen opens. There the user must select one topic, if none is selected and the user tries to progress, a similar toast message as before is shown. The user may add a new topic by pressing the orange button. Clicking on *Previous* return to the *Choose component type* screen. Pressing *Next* progresses to settings. Settings differ for each component, they change in the number and type of input fields. The user must fill out all input fields, otherwise he is met with error messages. Upon confirming, the new component is assed and the user returns to the dashboard. Users may return to the dashboard at any time by pressing the arrow in the top left corner.

Add topic screen

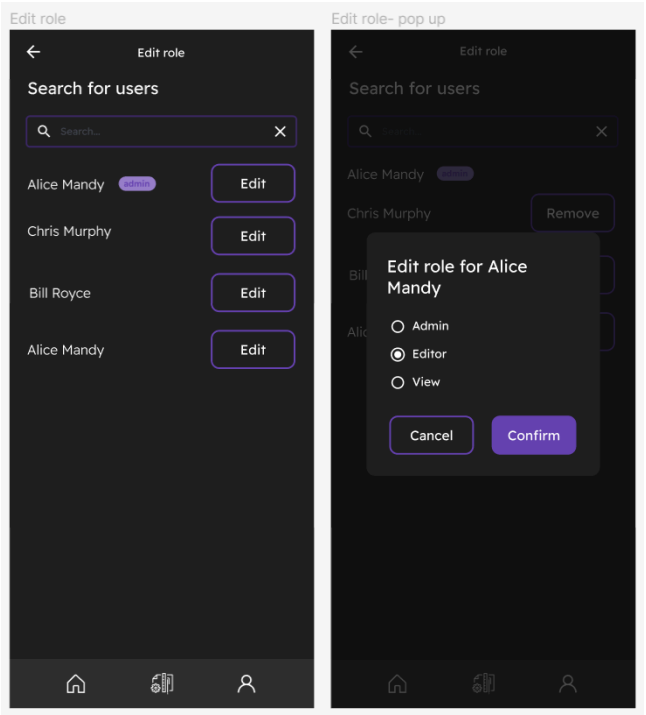
The add topic screen asks the user to input a new unique topic name and select from the list of data types. Upon confirmation, the topic is added and the user returns to the previous screen. In case of failure, a toast message is shown. If the chosen name already exists, the input field turns red and shows an appropriate message.

Fig. 22. The add topic screen mockups.



Search screen

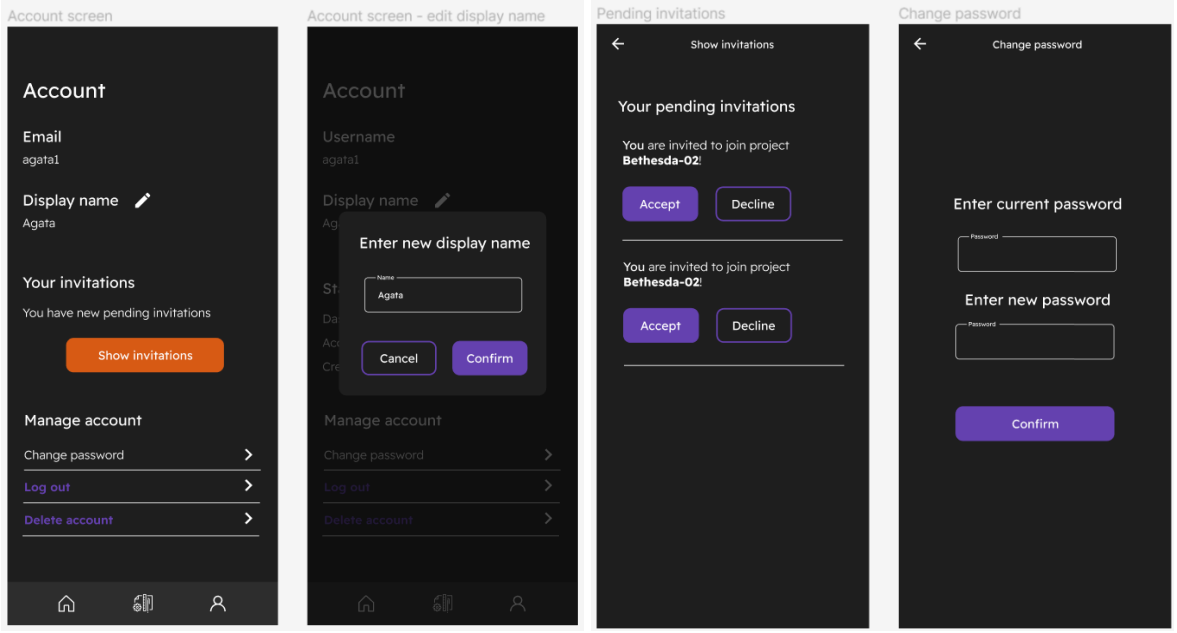
Fig. 23. The search screen mockups.



The search screen is made to be a reusable screen, as navigating from different points of the application opens this screen with slightly different settings. The button text, as well as the dialog, and even the list of users differ based on the screen type.
The user may search through the list of users by typing into the search field. Pressing the X in the search field clears the text. Pressing on the buttons located on the right starts an activity specific to the screen, usually opening a dialog.

Account screen

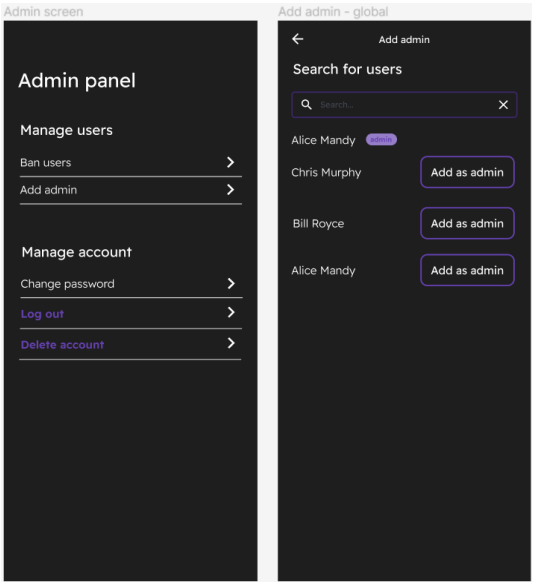
Fig. 24. The account screen mockups.



The account screen is a standard screen that allows the user to manage his own account. The user may change his display name by pressing the pen icon, he may log out or delete his account by pressing the menu items which open a dialog. The invitations can be entered from the account, where the user may decide whether to join a given project. Lastly, the user may change his password by clicking the menu item which navigates to the change password screen.

Admin screen

Fig. 25. The admin screen mockups.

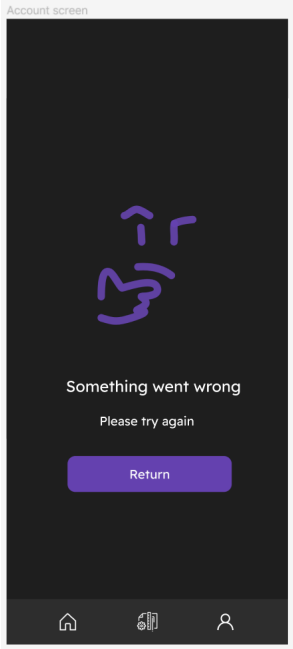


The admin panel is a very simple screen that consists of several menu items. As in an account screen, the admin user may log out, delete his account or change password. By

pressing the menu items in the *Manage users* section, the search screen is opened with appropriate settings.

Error screen

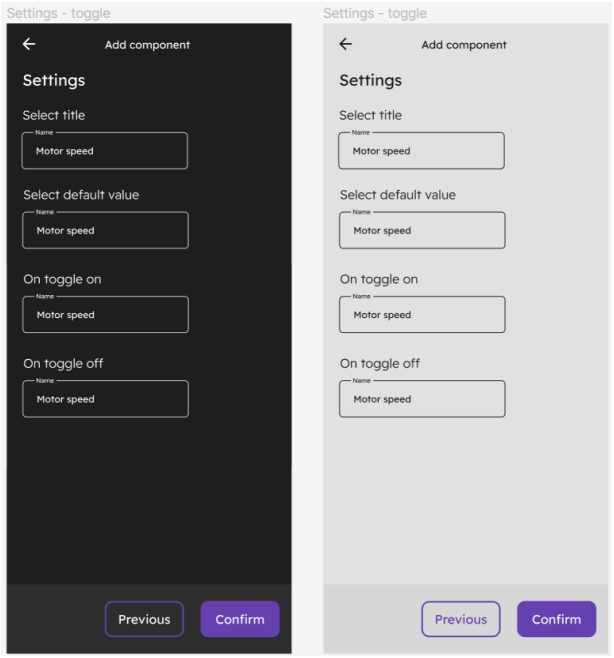
Fig. 26. The error screen mockups.



The error screen is a special type of screen which is shown throughout the whole app if any error occurs. It allows the user to return to a previous screen, if applicable.

Light and dark mode

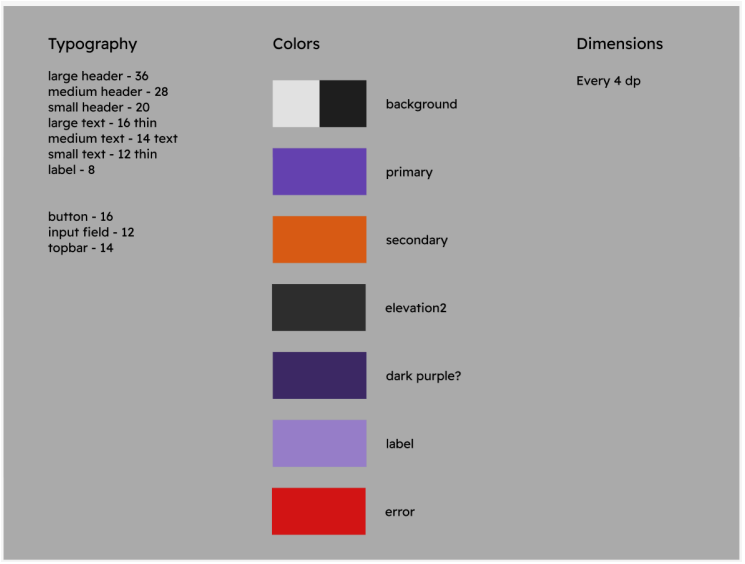
Fig. 27. The light and dark mode mockups.



Both light and dark modes have been prepared for this application, to allow the user to further personalize his experience, as well as increase the overall user satisfaction score. The primary and secondary colors have been kept the same, while the background and text colors have changed to reflect a good contrast.

Design scheme

Fig. 28. The design scheme of the application.



The design scheme contains all the colors, fonts and lists dimensions used throughout the app. Creating such a condensed schema simplifies the development process and helps keep the UI consistent.

8. Implementation

Backend

Containerization

We assumed three benefits coming from the project's containerization. We used Docker, and Docker Compose tool specifically, as this technology is able to meet our assumptions [37]. These three benefits are:

- Simplification of the installation and start up processes, as all dependencies are provided by Docker
- Running the Backend Server in a wide range of environments
- Unified management of configurations for all services inside the Backend Server

Both the development and deployment processes benefit from the containerization.

The implementation of containerization in our project consists basically from two parts:

- *Dockerfile*
- *docker-compose.yaml*

Dockerfile defines a multi-staged build for the Spring application. The idea of the multi-staged build is to avoid downloading dependencies by reusing the ones downloaded in previous builds, unless there were some changes in the pom.xml file. *Dockerfile* can be found in the Spring application's root directory.

docker-compose.yaml file can be found in the project's root directory. It is responsible for running multiple services that make up the Backend Server. There is a Spring application, RabbitMQ server, and a PostgreSQL database. The configuration properties and secrets are also automatically provided to all the services.

Security

Backend server is secured mostly using Spring Security but there is additional code on the level of the application logic that further controls the access to resources. In the implementation of the security mechanism using Spring Security, the following classes had to be implemented [38]:

- *SecurityConfig.kt* - configuration of Spring Security to adapt it to the project's needs. It defines the encryption used for password, the method of authentication, maps endpoints to user roles, and binds custom Spring Security class implementations to the overall security structure. ‘
- *UserDetailsImpl.kt* and *UserDetailsServiceImpl.kt* - implementations of UserDetails and UserDetailsService interfaces. These allow authenticating Users against the credentials saved in the Backend's Server database.
- *CustomAuthenticationFailureHandler.kt*, *CustomAuthenticationSuccessHandler.kt*, and *CustomLogoutSuccessHandler.kt* - custom handlers of login failure, login success, and logout success. These operations return appropriate status codes.

These implementations can be found in the *security* package.

Now, we will describe in more detail each important component of our implementation of security in the System.

Authentication
To authenticate, the User needs to send form data to /login endpoint. The Backend Server verifies the credentials against the Users saved in the persistent database (via the implementation of UserDetailsService). If the credentials were correct, a cookie with the session ID can be used to authenticate further requests. If /login fails, 401 status code is returned to the User.

Encryption
Utilizes BCryptPasswordEncoder as the password encoder, a widely used encryption algorithm for secure password hashing [39].

Authorization
There are two layers of authorization in the Backend Server: on the application level provided by Spring Security, and on the Project level.

On the application level, each request needs to be authorized. The privilege level necessary to access an endpoint is determined by the first part of its URI. There are three types of Users:

- anonymous - no authentication
- user - normally registered Users
- admins - Users with special privileges to manage accounts

On the Project level, the System checks if the currently logged in User has rights to perform a given action. These checks are done in the service layer of the Backend Server. There are four roles on the Project level:

- Project Owner
- Project Admin
- Project Editor
- Project Reader

The detailed privileges of these Project roles are defined in the use case specification earlier in the documentation.

REST API

Overall, there are 47 HTTP endpoints. They are grouped topically into controllers. Each group of endpoints corresponds to one type of resource. These groups are:

- Account
  Path pattern: /*/users/**
  Description: Everything related to account management. This includes actions available for Anonymous Users (account registration), actions for regular Users (changing account data), and actions for Admins (permission management).
- Project
  Path pattern: /user/project/**
  Description: Project management. These endpoints are available only for logged in Users. Project roles are often verified in processing of endpoints' logic.
- Topic
  Path pattern: /user/topic/**, /anon/topic/**
  Description: Topic management. Project roles are often verified in processing of endpoints' logic. There is one endpoint available for the IoT devices to retrieve Topic information. It requires the connection key of the corresponding Project.
- Dashboard
  Path pattern: /user/dashboard/**
  Description: Dashboard management. These endpoints are available only for logged in Users. Project roles are often verified in processing of endpoints' logic.
- Component
  Path pattern: /user/component/**
  Description: Just two endpoints for updating and retrieving Components for a given Dashboard.
- Message
  Path pattern: /user/message/**, /anon/message/**
  Description: Two endpoints for sending Messages, one for retrieving. The endpoint for Message retrieval allows for getting n last Messages for all Topics in a given Dashboard, which decreases the number of requests from the Mobile Apps. One of

the endpoints for sending Messages is provided for the IoT devices, authorization is conducted using the connection key inside the Message. This authorization process happens on the RabbitMQ server's level, which decreases the number of necessary database requests.

- Integration
  Path pattern: /user/integration/**, /anon/integration/**
  Description: All the endpoints necessary for handling external integrations. These include callbacks provided for external web services.

Synchronization

There are three processes that had to be synchronized between all Mobile App instances and IoT devices:

- Dashboard changes - every time there is a change in the Dashboard (layout change, or Component creation), all the Users currently interacting with the Dashboard need to see this change. These changes are transported to the interested parties with the use of WebSocket protocol (*ComponentChangeWebSocketHandler.kt*). Upon the establishment of WebSocket connection, it starts listening for changes using the *ComponentService.kt* Spring bean. Changes are delivered to this service by the listeners that intercept all the requests that may modify the Dashboard's contents (*listeners* package).
- Invitations - every time there is an invitation for the User, they should be instantaneously informed. These notifications are transported to the interested User with the use of WebSocket protocol (*InvitationAlertWebSocketHandler.kt*). Upon the establishment of WebSocket connection, it starts listening for invitations using the *InvitationAlertService.kt* Spring bean. Invitations are delivered to this service by the listener that intercepts all the requests that have to do with invitations (*InvitationRequestListener.kt*).
- Messages - when User is observing a Dashboard, they should be informed of any new Messages appearing on Topics that are used by Dashboard's Components. These Messages are transported to the interested User with the use of WebSocket protocol (*IncomingMessageWebSocketHandler.kt*). Upon the establishment of WebSocket connection, it connects to the RabbitMQ server, creates a new queue for every Topic, and binds it with RabbitMQ exchanges. The queue is removed after the connection is closed.

There are three techniques used in the implementation of these synchronization processes.

Request stream
Implementation of WebSocket handlers follows the concept of "client-request-a-server-stream" present in the RSocket protocol [23]. The client sends one message to the server upon the establishment of a connection. The message is used to determine what data is supposed to be streamed to the client. Then, the indefinite stream of messages starts flowing to the client.

Publish-Subscribe pattern
Messages stream follows publisher-subscriber pattern. This pattern is implemented by the usage of reactive programming techniques. It allows for setting up subscriptions that passively listen for any messages from the publishers. The publishers are the data sources [28].

Aspect oriented programming
We used AspectJ to establish the request interceptors. These interceptors are created declaratively in the *listeners* package. We can interpret them as declarative proxies. They are used to deliver data to the reactive publishers every time when the defined requests are processed.

Integrations

The System is integrated with external web and mobile services. The ones described in this section are integrations handled by the Trigger Components. These include:

- Gmail
- Discord
- Slack
- Telegram
- Mobile notifications

The implementation revolves around the *IntegrationManager.kt* class. It collects all the integrations defined in the database in the form of Trigger Components. For every Component, the connection to the RabbitMQ server is established, and a queue created. Every time there is a Message on one of these queues, the appropriate action is taken by the integration manager. The action to be taken is defined by the implementation of *IntegrationAction.kt* interface. There is one implementation per one integration. The instances of this type also depend on the configuration of corresponding Trigger Components.

Summary

The implementation provided for the Backend Server required more work than originally assumed. To outperform the quality requirements, the solutions had to be sophisticated and complex.

A lot of problems (especially synchronization) had to be solved using original approaches combining multiple techniques. These techniques included: aspect oriented programming, reactive programming, WebSocket handlers, and AMQP messaging. They exceed the scope of material taught during the studies.

Additionally, just the number of endpoints to be implemented was a challenge on its own.

Mobile Application

General

The IMCS Android application utilizes the design principles, mentioned in the Architecture part of this document, to create an easy to develop and maintain project structure. The code structure is divided into three main parts: data, model and UI.

The data package contains all of the classes used in data transmission, any Enum classes, result of API calls, and any classes used to hold information.

The model is used for data communication. It holds the local and remote data sources, as well as repositories that make the API calls, retrieve information and send it further down to View Models. This package also contains the WebSocket listeners. Each data source and repository is created using both an interface and implementation, to allow an easy stream of information to lower layers.

The UI package is the most extensive, as it contains all of the theming files, navigation, reusable components as well as screens along with their view models. This package is additionally grouped by features, to further divide the codebase into smaller components. Each package in the “features” package is a single screen of the mobile application, containing all of its visual elements as well as the logic regarding its state. This form of organization allows for easier maintenance of code, as well as improved scalability as a completely new feature can be quickly added on top of the existing application, completely separately from the rest of the code.

There are two additional packages – *app* and *helper*. App contains all the code used to initialize other frameworks such as Retrofit, as well as it holds the Koin module used for dependency injection. Helper is a space to keep all reusable functions that aid in writing various classes, e.g. Modifier extensions for writing Jetpack Compose functions.

The basic information exchange is as follows: the software makes API calls and retrieves information using the Retrofit framework in data source classes. The results from those calls are gathered and translated into readable results in the repository classes. Those classes are leveraged as interfaces to be used in the code of view model which gather results and transform the UI state of the application. Lastly, the UI layer observes the changes in the state and updates the visual components accordingly.

Data sources

The application contains two main sources of data – remote and local. Remote data access is conducted with the use of Retrofit and OkHttp frameworks which allow for sending network requests and receiving responses. The requests are structured as functions stored in an interface. They contain the path of the request, the method name of the API call as well as the desired response object. Such interfaces are then injected into repositories.

Additionally, Retrofit allows for adding custom interceptors, which was crucial in implementation of this project, as every API call required passing a session cookie to authorize with the server. In web based applications, this is usually done automatically by the web browser, however, in the case of mobile applications, this feature has to have been added manually.

Below is the code snippet of the custom interceptor.

Fig. 29. Code snippet of the cookie sessions interceptor.

```
Java
class AddCookiesInterceptor(
  private val userSessionLocalDataSource: UserSessionLocalDataSource
) : Interceptor {
  @Throws(IOException::class)
  override fun intercept(chain: Interceptor.Chain): Response {
    val builder: Request.Builder = chain.request().newBuilder()

    runBlocking {
      val activeUserCookie =
userSessionLocalDataSource.userSessionCookie.firstOrNull()
      activeUserCookie?.let {
        builder.addHeader("Cookie", it)
      }
    }

    return chain.proceed(builder.build())
  }
}
```

The session cookie is retrieved from the mobile phone's local storage which is passed as an interface into the interceptor using dependency injection.

Local data sources are the transactions with the mobile phones local storage. In the project, saving data to this storage has been implemented in two ways – using an app database by storing tables with records, and using Preferences and Proto DataStore.

Managing the local database is done with the help of Room library, which provides an abstraction layer over SQLite. It performs verification of SQLite queries, reduces error-prone boilerplate code, provides an annotation system, and streamlines migration paths [40]. Saving data in this method requires creating entities and Data Access Objects (DAO), which contain queries to the database.

The DataStore is a solution that allows for saving key-value pairs or typed objects, as an alternative to creating a whole new table in the database. This solution is optimal for individual records.

The Preferences DataStore stores and accesses data with the use of keys, however it does not provide type safety. The Proto DataStore provides type safety as it requires defining data types using protocol buffers, used for serializing structured data.

In the project, the Preferences data store has been used to save the current session cookie that must be sent with each request. This is a single value that requires no type check, therefore this was the most appropriate solution. On the other hand, the Proto DataStore has been used to save the logged in user data in order to limit unnecessary API calls to the backend server. In this case, the type of data should be consistent with requirements, therefore it is the more appropriate solution.

WebSocket listeners

WebSocket is an interface that enables full-duplex communication with a remote host [41]. The WebSocket listener is a special type of data source, specific for this application. It is a class that registers a new WebSocket on an injected client, and utilizes a callback function upon receiving new messages. Once the WebSocket begins a connection, it keeps on listening to incoming messages until it is closed manually using the *closeWebSocket()* function. Therefore, it is crucial to remember to close the connection upon e.g. leaving the view model so that the unused flow of data does not clutter the network.

Fig. 30. Code snippet of the invitation WebSocket listener.

```Java
class InvitationAlertWebSocketListener(
  client: OkHttpClient,
  onNewInvitation: (data: Boolean) -> Unit
) {
  private val request = Request.Builder()
    .url("ws://${BuildConfig.APP_NETWORK}/invitations")
    .build()

  private val webSocket = client.newWebSocket(
      request,
      object : WebSocketListener() {
    override fun onOpen(webSocket: WebSocket, response: Response) {
      Log.d("Invitation", "onOpen WebSocket called")
      // WebSocket connection is established
      webSocket.send("")
    }
      ...
  )
}
```

In the project, the WebSocket listeners are usually started upon creating a view model, however with the exception of invitations and notification, which need to be kept alive throughout the whole time of using the application by a user.

The callback functions are, therefore, usually defined in view models, and they handle UI state updates based on the message received.

A crucial use case of this technology for our application is receiving incoming messages from the IoT device. This protocol enables the application to register when a new message has been received, and update the UI state of the dashboard to reflect that change. In this way, the system performs real communication of the mobile application with an IoT device.

Repositories

The repository pattern is widely used in various fields and applications. As described in the *Domain Driven Design* book by Eric Evans, “Repository provides a simple interface to receive domain objects and manage their lifecycle”. The responsibility of the repository is to gather domain objects through interfaces of the data sources, and manage them by updating, deleting, adding or transforming, so that they can be used in the lower layers of the system. This pattern follows the **clean architecture** mentioned in the Architecture chapter of this documentation – it serves as the business layer, also called the domain layer.

In the project, the repositories have been created as interfaces with separate implementations, which allows for injecting them into view models using the Koin framework. Repositories utilize Enum classes to send meaningful data to the view model instead of API result codes which often vary in meaning.

Features

Features are the core part of the Android application. They represent the UI views along with their view model which store and modify their state. Each feature is a separate screen of the application. All of the screens work as separate modules, they do not depend on each other. Communication between the screens is handled using **navigation**, using which data can be passed in a form similar to web page paths.

Each screen is based on the MVVM design pattern which separates the code of the UI and the business logic. The UI state of the screen is held in a separate class which is modified in the view model, and observed on the UI using a shared flow. The user interaction functionality is defined in the view model, and it is accessed by the UI view using an interface.

The view models receive repositories through dependency injection to connect with the local and remote data sources. It is important to note that each function call of the repository is surrounded with a try catch block in the view model to handle exceptions arising from the database and remote server, as well as any connection issues. This is performed on the

view model so that it can easily manage the state of the UI and inform the user of any problems.

Another important note is that throughout the app have been implemented special error and loading screens to make the experience seamless for the end user. Any input fields contain special error messages and turn red upon entering an incorrect value, informing the user of the problem.

Toast messages have been also implemented to inform the user of a successful or failed event. They are sent from the view model and received on the UI view, as they require application context to activate. In order to achieve this, a special class has been implemented in the *helpers* package called Toast which emits a String message from the view model using a shared flow, and exposes a *collect()* function to the UI to collect the passed event and show the message to the user. This sort of structure is required for actions that originate from the view model but the UI should perform [42].

Another class Event has been implemented in a similar way as a more generic event for different types of tasks, mainly performing navigation upon receiving a certain result from the backend server.

Fig. 31. Code snippet of the custom Event implementation.

```
Java
class EventImpl : Event {
  private val _event = MutableSharedFlow<Any>()

  override suspend fun event(message: Any) {
    _event.emit(message)
  }

  @Composable
  override fun CollectEvent(context: Context, callback: (it: Any) -> Unit) {
    LaunchedEffect(Unit) {
      _event.collect {
        callback(it)
      }
    }
  }
}
```

1. Add component screen

The purpose of this screen is to allow the user to create a new component on the dashboard. It has a special structure, as the user needs to fill out several steps in order to complete the process. The steps are: selecting a component type, selecting a topic, and filling out required settings. Some components, such as Discord, have an additional last page for special functionality, e.g. choosing a Discord channel.

In order to manage those steps on a single view without traditional navigation, a separate mock navigation has been created. It is simply a state called *currentPage* kept on the view model which stores the current page type in order to inform the view which components should be shown. Upon navigation to a next or previous step by the user, the functions *navigateNext* and *navigatePrevious* are called respectively, which run the appropriate code based on the currently saved page type and update the view.

The AddComponentViewModel class is extensive, as it needs to generate appropriate data for the component chosen by the user. Components have various requirements that need to be fulfilled in order for a successful creation. Certain components require a specific type of data, e.g. the slider requires numeric data, which is why the topic list in the second step is being filtered based on the component type in the function *getFilteredTopics()*.

The settings input fields also depend on the chosen component type, therefore, in order to simplify this process and separate the business logic from the UI, the list of required fields is generated in the view model class, and the UI view only iterates through the incoming input field list and visualizes the data passed.

To be precise, the exposed state is a **map** of input field type and input field data. This has been done in order for quick and easy access of the data when creating a Data Transfer Object (DTO) to be sent to the backend server. This solution has been reused throughout the application in many screens that utilize a list of input fields.

Another important part of this feature is the handling of the Discord component. This is a special component, as it contains an additional screen, in contrast to other components. Upon filling out the settings and navigating to the next screen, a new **intent** is created which opens the web browser. There, the user must authenticate and choose the server, and upon returning to the application choose a discord channel from a generated list.

Lastly, this feature utilizes an important functionality of the Koin framework – namely navigation using Scope. From the Koin documentation, “Scope is a fixed duration of time or method calls in which an object exists” [43]. Using scopes allows one to create and pass an object between screens, in contrast to the typical way of passing information in the paths of screens, which forces the data to be in a String form.

The AddComponentViewModel requires receiving the list of all current components in order to update it with the newly created component. This is forced by the design of the backend implementation.

2. Add topic screen

The role of this screen is to allow the user to create a new topic. This feature is accessed from various points in the application, which is why it needs to be a separate screen. The content consists of a single input field for the name of the topic, and a radio button list of topic data types. Such a list allows the user only to select a single item.

The name input field contains a special error message that warns the user if the inputted name already exists in the project. The name of the topic must be unique.

The screen provides descriptions to explain to users the need for such information.

It is important to note about the navigation between this screen and the project details. Normally, the function *popBackStack()* used for navigating back throughout the application hierarchy of accessed screens does not invoke a recomposition. This means that after creating a new topic, the user would come back to a non-updated screen, and wonder where the new item is. To conquer this issue, a special parameter is passed along in the *previousBackStackEntry* to inform the previous screen there has been a change and the view must update.

Fig. 32. Code snippet of passing the *previousBackStackEntry* parameter to the previous screen.

```
Java
navController.previousBackStackEntry?.savedStateHandle?.set(
      "resultStatus",
      true
)
```

3. Dashboard screen

Overview

Dashboard screen contains the core functionality of the application. It is the view which holds the components or interfaces that allow for communication with the IoT device.

This screen is both simple, as it only contains a list of components, and complex because of the nature of the components, and the ability of the user to control the list.

Dashboard modes

Three different modes of the dashboard can be distinguished.

The first is the default view for the user with the Viewer project role. Such a user can only interact with components, but cannot change them in any way. In detail, the Viewer user cannot add new components to the dashboard, and cannot reorder or delete them. He may only interact with the functionality of the components, e.g. pressing a button.

The next mode is the default screen for the Editor and Admin users who may change the state of the dashboard screen. They may add, reorder and delete the components. On the

default screen, however, the users are allowed to navigate in order to add a new component, or interact with the functionality of the interfaces.

In order to modify the list of components, another mode was created, called the Edit mode. In this view, which is accessed through a menu in the top bar of the screen, the user can rearrange the components, as well as delete them.

To create such a varying interface, the view model initially identifies the user by his role in the project, and an appropriate view is built upon this information.

View model initialization

To correctly set up the screen, upon entering the initialization of the view model, several important steps occur.

Firstly, the dashboard is saved to a local database to enable faster access for users from the main screen of the application. A *DashboardEntity* is created and passed to the *saveLastAccessedDashboard* function of the Dashboard Repository. This function first searches for an existing dashboard in the local database with the same id as the current one, deletes this entry and enters a new record with the current data. The process is programmed in this way, so that the last accessed dashboard is always at the bottom of the database record list, meaning it has the largest id. This is necessary in order to manage the order of the accessed dashboards for listing on the main screen.

The next important process in the dashboard view model initialization is creating the *ComponentChangeWebSocketListener* object. This is the WebSocket listener that listens for changes in the component list on the dashboard, such as reordering the list or deleting an element. If any class of this has been instantiated before, its connection must be closed before creating a new one.

The *onComponentChangeMessage* callback function is passed when instantiating this class. This function sorts the received list by index, as this parameter resembles the placement of the item in the list in the backend server database, then it redefines the *MessageReceivedWebSocketListener* in case there have been added components that connect to new topics, in which case their connection needs to be established in order to listen to their changes, and lastly it updates the UI state with the new component list along with its last received value.

Fig. 33. The code snippet of the callback function for component list change.

```
Java
private fun onComponentChangeMessage(data: ComponentListDto) {
    _componentListDto = _componentListDto?.copy(
      components = data.components.sortedBy { it.index }
    )

    val topics = _componentListDto?.components?.mapNotNull {
it.topic?.uniqueName }
```

```
    topics?.let {
      messageReceivedListener?.closeWebSocket()

      if (it.isNotEmpty()) {
        messageReceivedListener = MessageReceivedWebSocketListener(
          client = client,
          topicNames = it,
          onMessageReceived = { m -> onMessageReceived(m) }
        )
      }
    }

    _uiState.update { ui ->
       ...
    }
  }
```

This WebSocket listener is the crucial component of **synchronization** in the project. It allows users from different accounts to see changes on a dashboard in real-time. This allows users to collaborate with each other on projects, as every change is reflected on all devices with access.

Next in the view model initialization is a section of code which is run in a Coroutine as it performs API calls from the repository. The state is updated to notify about loading, as the user must wait for those actions to finish in order to make every functionality on the screen accessible. The current component list is fetched from the backend server, and the *MessageReceivedWebSocketListener* is initialized with the topics used by those components. This WebSocket listener is responsible for listening on incoming messages from the IoT device – it is therefore the core functionality of our application. The *onMessageReceived* callback function updates the *_lastMessages* field of the view model class by either appending the newly received message to an existing topic, or in case a new topic has been received – appends a new object containing the topic with the received message. Then, the UI state of the components is updated to reflect the new data visually.

Fig. 34. Code snippet of the callback for a new message received.

```
Java
 private fun onMessageReceived(data: MessageDto) {
    val topicId = data.topic.id ?: return
    val currentMessages = _lastMessages
    val newMessages = if (topicId !in currentMessages.map { it.topicId }) {
      currentMessages + listOf(
        TopicMessagesDto(
          topicId = topicId,
          messages = listOf(data)
        )
```

```
            )
        } else {
            currentMessages
                .map {
                    if (it.topicId == topicId)
                        it.copy(
                            messages = it.messages + listOf(data)
                        )
                    else it
                }
        }
        _lastMessages = newMessages

        _uiState.update { ui ->
            ...
        }
    }
```

Lastly, in the view model initialization, the UI screen is updated to reflect the gathered component and user data, as well as loading is set back to false to enable the user to use the screen.

Components

The main functionality of the dashboard screen, as well as the core of our application, are the components which serve as an interface of communication of the mobile application with an external IoT device.

There are in total 12 different components, which all possess distinct functionality. In the project, they are distinguished and identified using the *ComponentDetailedType* Enum class. Additionally, the components can be divided into 3 larger groups – Input, Output and Trigger, which possess another Enum class called *ComponentType*. This distinction is based on the purpose of the components. The Input components are such that take input from the user, and send data to the IoT device, e.g. a button. Output Components are those which take data from the IoT device and present it to the user in the form of graphs. Lastly, the Trigger components are a special type that refers to integration of the application with external services, e.g. sending messages to a Discord channel.

Each time has its own name, index representing its position in the list, size and height (note there are only two sizes – half of width of the screen and full width, while the height can be any value), type, assigned topic, values to send if applicable, and the current value from the topic (or values for graphs). Additionally, each holds an *absolutePosition* attribute which is used when reordering the item.

Upon clicking a component, the *onComponentClick* function is called which performs an appropriate action based on the component type. In general, it chooses the appropriate message to send, creates a new *MessageDto* object, and then sends the message to the given topic using *MessageRepository*.

Below you can find the full list of each component's functionality and behavior.

1. Button component.

The Button is a simple component which sends its *onSendValue* defined in the settings upon being pressed. It can only send one message, its value is always the same.

2. Toggle component.

The Toggle is a more complex component as it possesses two states: on and off. Depending on those states, it sends a different value to the topic, *onSendValue* and *onSendAlternative* respectively. The component is on when the current value of the topic is equal to the *onSendValue*, and off otherwise. Upon pressing, the UI state is updated locally to reflect a quick change for the user, as the message is sent to the topic in the background.

3. Slider component.

The Slider is another input component, which allows the user to choose a value from a selected range. It does not utilize the *onSendValue* and *onSendAlternative* fields but rather focuses on the minimum and maximum values. This component only accepts numeric data, which is checked when adding this component to the list. As the user selects a new value, the UI state is updated only locally to reflect the movement that the user makes. Upon release, the new value from the range on the slider is sent to the topic.

4. Release button component.

Release button is an additional component which was not planned for. The need for such a component appeared when creating a demo for this project. This type of component sends out two values, *onSendValue* and *onSendAlternative*. Upon clicking and holding the button, the *onSendValue* is sent, and when the button is released the *onSendAlternative* is sent instead. This component works well for functionalities which require active user input while they should be performed, as well as a stopping signal which notifies when the user wants to stop the action.

5. Photo component.

The Photo component is the last of the Input component. When a user pressed this kind of component, a new intent is created and started which leads the user to the native camera app of the phone. The user may take a photograph, and upon confirmation, returns back to the IMCS application. The intent carries data from the camera application with the bitmap of the taken photograph. In the function *onTakePhoto* this bitmap image is compressed, translated into a byte array, and encoded using Base64 in order to be transferred across the network. Such an image can be easily received on an IoT device and shown e.g. on a LED display. This component requires data of type Image.

6. Line graph component.

The Line graph is an Output component. The component had been drawn using a native Canvas by manually drawing each line with Bezier curves. It shows the last set of received data on a time scale shown in hours. The minimum and maximum values of both graph axis needed to be found in order for showcasing data in a proper format. When a new message is received by the view model, the change is reflected on the graph by adding it to the set of visible values. The graph component accepts only numeric data.

7. Speed graph component.

Speed graph is another Output component, which shows the current value of a topic on a chosen scale. This component utilizes the *min* and *max* fields in the component class, chosen in settings when adding the component, to create a range of values, and shows the current value using a “needle” that points to the correct value in the range. This graph accepts only numeric data.

8. Discord component.

This component is a Trigger component. It integrates the application with the Discord software to enable users to send messages when the chosen topic has received a new message. Setting this component requires an additional step compared to other components, as it asks the user to authorize with their Discord account and select an appropriate channel to send messages to. This component works in the background, only listening to the incoming messages, the user does not need to press anything to activate it.

9. Email component.

Email is another Trigger component, which allows the user to send an email upon receiving a new message by a topic. The email is sent to each person with access to the project. The user may select the title and content of the message. Similarly to other Trigger components, it works in the background and does not require any input from the user.

10. Slack component.

This component works very similarly to the Discord component, except in order to set up, it requires more input in settings from the user. It allows sending a message to a chosen channel, including the value of the message that notified the component to transfer a message to the external service.

11. Telegram component.

Telegram works very similarly to the Slack and Discord components, it allows the user to send messages to a Telegram channel.

12. Notification component.

This is another Trigger component which utilizes the phone’s native notification system to inform users of new messages in a topic. It’s behavior is similar to other trigger components, as the user needs to set up a message to show upon receiving a new message, however, it

requires a significant change in initialization. Notification should be received on the phone of the user even if the application is completely closed. Therefore, it is necessary to start this process as a **background service** in Android. From the Android documentation, a service is a component used to perform longer-running actions without interacting with the user or extending functionality for other applications [44]. This service uses a WebSocket listener called *NotificationWebSocketListener* to listen to incoming notifications from the backend. It is started when the user logs into his account, and is kept alive as long as the user is logged in.

An important note about this component is that it requires appropriate permissions from the user in order to show notifications. For Android phones with SDK above 33, the permission is not set b y default, so the application must show a dialog asking the user for permission. This functionality is held in the *PermissionRequest* composable function. It uses a result launcher and *RequestPermission* contract to ask the user for permission and return a result into the application.

Component list reordering

Another important aspect of the dashboard screen is the reordering of component items in the list. The view itself is actually a lazy grid, as it draws the components in both horizontal and vertical direction, and only memorizes those which are currently visible. In order to allow for efficient reordering together with synchronization with different users, the order of the list is kept using the *index* field in each component. In this way, the order can be easily stored and sent to the backend.

The list can be reorder while in edit mode, the user must long press the desired component, and then he may move the item around the screen. The actual position of the item on the screen is being constantly calculated and stored in the *absolutePosition* of the component. The value is taken from the *onGloballyPositioned* modifier that is kept in the component view wrapper. Upon placing the component, the view model function *onPlaceDraggedComponent* is called, which, based on the absolute position of the component on the screen, calculates the index of the component which is closest to this position. It does so by using the information on the currently visible items from the *LazyStaggeredGrid*. Once the new index is known, the indices of each component are updated, and the list is sent to the backend to allow for seamless synchronization of the dashboard with other users. The change in component order is animated in the list.

Fig. 35. Code snippet for finding the index of the item which is closest to the current position of the dragged component.

```java
for (it in visibleItems.subList(toSubtract, visibleItems.size - 1)) {
      val currentItemIndex = it.index-toSubtract
      // Do not consider the original position of the currently dragged item.
```

```
        if (currentItemIndex == itemIndex) continue

        // Calculate the current distance squared.
        val currComponent = components.getOrNull(currentItemIndex) ?: return null
        val width = if (draggedComponent.size == 2) windowWidth else windowWidth/2
        val draggedCenterPos = draggedComponent.absolutePosition + Offset(width/2,
draggedComponent.height.value/2)
        val currCenterPos = currComponent.absolutePosition + Offset(width/2,
currComponent.height.value/2)
        val currDiff = (draggedCenterPos - currCenterPos).getDistanceSquared()

        // If the new distance is smaller than currently saved, assign it
together with the closest index.
        if (currDiff < diff) {
          diff = currDiff
          closestIndex = currentItemIndex
        }
}
```

1. Invitations screen

Invitations is another functionality of the application that requires careful consideration. The user should be notified of receiving a new invitation while using the application, therefore, the system should constantly listen to new invitations received. When a new invitation has arrived, the application shows a toast message to inform the user, and a red dot is placed on the bottom navigation on the Account tab, as this is where all pending invitations are kept.

Background listener for the invitations has been implemented with the use of WebSocket in *InvitationAlertWebSocketListener*. The code which sets up this listener is in the **onCreate** function, as the connection needs to be established upon creation of the application, and used all the time throughout the lifecycle. It is important to note that, additionally, the code listens for changes in logged in users, as in order to connect, the user must be authorized with the system. The code is performed on a background thread so as to not disrupt the process of creating the application.

Fig. 36. Code snippet of creating the invitation WebSocket connection on a background thread.

```
Java
CoroutineScope(Dispatchers.Default).launch {
        userRepository.getLoggedInUser().collect {
```

```
        if (it != null) {
          invitationAlertWebSocketListener?.closeWebSocket()
          invitationAlertWebSocketListener = InvitationAlertWebSocketListener(
            client = client,
            onNewInvitation = { data -> onNewInvitation(data) }
          )
        }
       ...
    }
}
```

2. Learn screen.

Learn is a very simple but very important screen that serves as a tutorial for the user to get to know the application. It is a static screen, as it contains no dynamic elements, except for a button that navigates to the Projects screen. It is a list of texts that explain the process present in the app, and what are the first steps the user should take after downloading. This screen can be accessed from the main screen of the application, so all new users can easily find it and learn more about the app. Due to the lack of dynamic elements and therefore lack of screen UI state, this screen requires no view model.

3. Login and register screens.

This set of screens enables the user to authorize and personalize their experience with the IMCS system. Such features are very common in most applications, however one should note that such processes should bring convenience to the user and the system, otherwise they should be deemed unnecessary [45].

The set of processes follow the industry standard. There is a login page where the user may insert their username and password, a registration page where the user can set up a new account, and a forgot password functionality which enables the user to retrieve password if needed. Additionally, there is a requirement from the user to activate his account before being allowed to login.

The full process is as follows. The user navigates to create an account. He must enter a valid email, name and password. The user is notified of any mistakes in the input fields. Upon accepting his data, the user is navigated to activate his account, where he must enter a code received from email. Lastly, the user is sent back to the login page, where he may be authorized to use his newly created account.

The screens along with their view models are gathered into a single package to modularize the whole process. The screens do not make sense without each other – a user without an account cannot login, he must be able to create a new account. Retrieving a forgotten password only makes sense when the user is trying to input a password, therefore, it must

be located on the login page. Each of the screens needs direct access with one another, therefore, they are in a way co-dependent. However, they all constitute a separate process that does not need to interact with any other part of the application. Therefore, it can be stored as a separate module.

4. Main screen.

Main screen is the first screen that the user sees upon logging in. It may not contain any core functionality for the application, but in terms of user experience, it is one of the most important screens. This is the space that should welcome the new user into the application and encourage him to use the app further, as well as make it convenient for a long-term user to navigate to important functionalities of the app. It is also a good space to inform the user of any changes, as he will always stumble upon this information.

This screen varies based on the collected data about the user. For a completely new account, this screen focuses on navigating the user to create a new project and get started with the core functionality, as well as emphasizes the learning process by giving access to a tutorial.

For a user that has already been using this application, this view presents a list of last accessed dashboards, in reverse order of accessing them. In order to showcase such a list, the app stores information about the accessed dashboards in a local database. It keeps the ids of dashboards as well as the ids of users who accessed them to show appropriate information to the active user. In this way, the user can quickly return to previous work upon opening the application.

Additionally, if the WebSocket listener registered any new invitations, this screen shows a section that informs the user about this and allows to navigate to the screen where he can make a decision about the invitation.

Lastly, this view launches the permission dialog mentioned before, to ask the user for notification permissions.

5. Project details screen.

Project details is another view that is core to the functionality of the system. It is the space where all of the dashboards, topics, and group members can be modified and accessed. Due to the large amount of information that is required to be shown, the view has been programmed in a special way. It uses tabs to separate the three groups of information, in order to declutter the amount of data and enhance the user experience.

In order to navigate between the tabs, the currently selected tab is kept as state on the view model. The appropriate view is drawn based on this information.

There is an additional complexity introduced for this feature – the users have different roles in the projects, and therefore, have different access to various functionalities. The Viewer user can only access the dashboards, topics and group members, but cannot modify anything. The Modifier can add new items, but may not affect the other members. The

Admin is allowed to modify everything in the project, as well as delete it and modify the status of other group members.

In order to take those roles into consideration, some fields must be generated by the view model, such as the group user options. They are dynamically created based on the role of the user, and the UI simply draws what generic items it receives. Some functionalities, such as buttons for adding new items, must be hidden for certain users, which must be done on the UI of the screen.

Lastly, specific behavior for when the project is deleted must have been implemented, as some users may still be using the project while it has been deleted. To handle such a case, upon entering this view, the view model initializes a WebSocket connection that listens for project deletion. Upon receiving such a message, the user is informed and navigated back to the Projects screen. The WebSocket connection must be closed upon destruction of the view model.

6. Group screen

The Group Screen provides an overview of the current project user group. Some of the things included there are role descriptions and a list of members. Standard users have the option to leave the group, while administrators have additional functionalities, such as inviting users, editing roles, revoking access, and adding new administrators. Every option is straightforward and aims for efficient management of the group for administrators.

7. Projects screen.

This screen is a simple list of all the projects the user has access to. It contains a dialog window that allows the user to create new projects. It is important to note that the project name must be unique – the user is informed if he enters an invalid name.

8. Account screen

The user settings panel offers a range of options for users to customize their experience.

Users are provided with their email address, and displayed name with the option to change it which allows users to personalize how they appear on the platform. The Invitations section offers a list of received invitations to other user projects.

Under Manage Account, users have the option to change passwords, enhancing account security. The Log Out option as the name suggests provides a simple way to securely log out of the current account, while the Delete Account feature facilitates a straightforward account deletion process for those who want to delete their account.

9. Admin screen

The admin settings panel empowers administrators with tools for efficient account management.

Within Manage Users, administrators have the authority to Ban Users, adding an extra layer of control and security. The Add Admin option adds the possibility of the process of designating additional administrators, increasing overall management capabilities.

Similar to the user settings, administrators also have access to Manage Account options as base functionalities.

This design ensures that users and administrators will effortlessly navigate through the settings panel managing their accounts with efficiency and without problems.

10. Search screen

Our Search Screen contains various functionalities such as inviting, banning, and role changes with seamless efficiency.

Additionally, this solution introduces instant search capability. Users no longer need to type an entire phrase and then initiate a search. Instead, the search options dynamically narrow down with each typed letter. This solution provides a good user experience and overall user satisfaction.

To eliminate any user error, every action undertaken by a user triggers a specific confirmation dialog. This additional layer of confirmation serves to prevent accidental clicks and increase confidence in users.

For some search options like banning users, it is important to take into account users' current status. For instance, if a user is banned, the search screen dynamically displays the unbanning option, and vice versa.

To ensure that users cannot e.g. ban themselves, change their status, add an admin role to a user who is already an admin, or invite users to a group who are in the group at the moment we introduced filtering based on the desired process. This filtering mechanism presents users with a refined list adjusted to their specific needs.

The search screen has been created in a generic way, so that it can be reused with varying functionality, as the screen is called from many areas of the application and requires different actions. In order to achieve this, while navigating the screen must receive a type parameter, which determines the content of the screen. The varying UI and functionality is defined on the view model so that it can be easily updated and observed on the UI view.

11. Main activity.

Jetpack Compose follows a single activity architecture. It has been measured that advanced android applications with multiple activities have increased energy consumption over single activity applications [46]. Such an architecture additionally decreases the complexity of the code, making it easier to develop and manage.

In the project, the *MainActivity* is used mostly to set up the Jetpack Compose content. In the *onCreate* function, the app theme is defined, and the starting views created. Additionally, this function runs the code for setting up the invitations listener and notification service, as those processes need to be constantly in the background of the app.

An important fact about this *AppContent* is that it decides what screen should be first opened based on the user state. If the user is logged out, the login page is shown, if the user is an admin, he is navigated to the admin screen. Otherwise, the default main screen is opened.

Additionally, this is the place where bottom navigation is set up. The Android documentation informs us that the bottom navigation bar allows the user to switch to different views easily, and makes the user aware of the most important screens in the application [47]. This component also initializes the entire navigation for the application, defines the starting route, and stores information which screens need the bottom navigation hidden.

Theme

The application uses Google's Material Design 3 which provides a standardized way of defining and usage of colors, typography and shapes. The theme allows the application to keep a consistent design through every screen, and greatly simplifies the development process [48]. It can be separated into 3 components – the colors, typography and dimensions. Theme is what groups those elements and makes them easily accessible while developing the UI.

The IMCS application utilizes both dark and light modes of the application, which requires setting up two different theme color schemes to be used appropriately. Thanks to this change, the user experience can improve greatly, as users vary greatly in terms of their preferences.

Summary

The implementation of the Mobile app required using many non-trivial patterns and architectures. To implement all of the features created on the backend, the code of the Mobile app had to be very extensive. Each feature requires to be surrounded with a detailed layer of UI, which allows the User to easily and intuitively use the given functionality. Implementing this amount of features proved to be a great challenge, especially to keep the quality and performance of the application on a high level.

The creation of the Mobile application exceeded the scope of studies from our university, as it required extensive knowledge of reactive programming, using the WebSocket communication protocol, and vast knowledge of the newest technologies in Android mobile development.

9. Tests/Results and discussion

Discussion of the project's results, including tests and the example of the application. Tests and their results are cross referenced to functional and non-functional requirements. These references appear in the following format: *[NFR X]*, where X is the requirement number.

Backend

Unit tests

Frameworks and methodologies

MockK
Mocking library used to create mocks of classes and interfaces, allowing simulated behavior in tests.

JUnit 5
Testing framework for Java and Kotlin used for writing and running tests.

Given-When-Then Structure
Tests are structured in a "given-when-then" format, where specific conditions are set up, actions are performed, and outcomes are asserted.

Unit tests were prepared for all services containing more complicated logic. They cover positive and negative scenarios including throwing exceptions. Scopes of the unit tests with examples of chosen test cases are provided below.

Component Service Test [FR 3] [FR 7] [FR 8] [FR 10]
updateAllPositiveCase:
Validates the update functionality of multiple components. This relates to the requirement of managing Components on Dashboards to monitor and exchange data between the Mobile App and IoT Devices.

updateAllUserNotAuthenticated: Tests the case when a user is not authenticated, ensuring that unauthorized access is restricted, aligning with the user authentication requirement.

Dashboard Service Test [FR 2]
createDashboardPositive: Verifies the creation of a new dashboard, fulfilling the requirement of creating and managing personalized Dashboards.

Message Service Test [FR 1] [FR 6] [FR 11]
sendMessagePositive: Tests the successful sending of a message, aligning with the requirement of exchanging data between the System and IoT Devices remotely.

getLastMessagesForDashboardPositive: Validates fetching the last messages for a dashboard, which aligns with the need to view recent data from IoT Devices.

Project Service Test [FR 5] [FR 12]
revokeAccessPositive: Validates the revoking of user access to a project, aligning with the requirement to manage project access and privileges for users.

editRolePositive: Ensures the ability to modify user roles within a project, which is tied to managing privileges of project users.

createInvitationPositive: Verifies the creation of invitations to join a project, linked to the requirement of allowing users to share their projects with others.

Reset Password Token Service Test [FR 13]
createVerificationTokenTest: Verifies the creation of a token for resetting passwords, aligning with the requirement of enabling users to reset their passwords securely.

Topic Service Test [FR 4]
createTopicPositive: Tests the creation of topics within a project, aligning with the requirement for users to define topics to establish communication links between the Mobile App and IoT Devices.

deleteTopicPositive: Validates the deletion of topics, which relates to the requirement of managing and removing topics when needed.

User Service Test [FR 13] [FR 15] [FR 16]
createUser: Tests the creation of users with admin and user roles, hashing passwords, and saving them correctly.

verifyUser: Validates the user verification process and its handling of active and inactive users.

toggleBlockedById and changeUserRole: Tests for handling user blocking and role changes.

Verification Token Service Test [FR 12]
createVerificationTokenTest and findActiveByCode: These tests are associated with token generation, validation, and ensuring tokens are properly managed with expiration and usage constraints.

Load and functional tests

To verify non-functional requirements on the backend side of the System, the load tests are provided. By exposing the System to the conditions of normal and extreme usages, its performance and reliability are verified [49]. These tests can be found inside the *backend/tests* directory. There are also functional tests that check the validity of particular Backend Server's features [50].

Tests are grouped into files by their scopes:

Account management tests [NFR 4] [NFR 5]
Tests in this scope focus on data consistency, and unauthorized access. These are functional tests. Different requests trying to break the data consistency, or to access resources without necessary authorization.

Result: No errors.

API tests [NFR 2] [NFR 3]
Tests in this scope focus on performance and reliability of the REST API. These are load tests. 1000 requests at once are sent to the endpoints that utilize a lot of system resources (modifying tables with multiple constraints).

Results: Average response time below 1 second. No errors.

Synchronization tests [NFR 2] [NFR 3]
Tests in this scope focus on performance and reliability of the synchronization mechanisms. These are load tests. There are 50 WebSocket connections established listening for the changes in the same Dashboard.

Results: All the updates are delivered in less than 1 second. 0% data loss.

Mobile Application

Unit tests

Frameworks

MockK
Mocking library used to create mocks of classes and interfaces, allowing simulated behavior in tests.

JUnit 4
Testing framework for Java and Kotlin used for writing and running tests.

Methodology

The unit tests were performed in a "given-when-then" format, similarly to the backend tests. The tests were performed on the repository and view model classes to verify the correct and false scenarios, and ensure the stability of results for future changes in code.

Performed tests

1. Repositories

Each repository test class contains several tests which check whether the repository returns proper results for the appropriate API call results. All of the test classes follow the same

structure. Initially, the mocks of data sources used by the repository are created using the *mockk* function from Mockito. The mocks are inputted into a new instance of the tested repository. Then, the proper functionality of the data sources must be mocked so that when the test calls the given function from the data source, the function returns the desired result we set instead of using the real implementation.

Fig. 37. Code snippet of an example of data source mock function.

```
Java
private fun everyCreateDashboardReturnsFailure() {
    val responseBody = "operation failed".toResponseBody()
    val response = Response.error<DashboardDto>(400, responseBody)

    every {
      runBlocking { remoteDataSource.createDashboard(any()) }
    } returns response
 }
```

Then, when all needed functions are mocked, the test cases may be created. Each function from the repository is tested for several scenarios depending on the type of result returned by the data source functions. A separate test is created to verify the program result for each handled result code from the backend server.

The tests assert whether a checked value is equal to what we expect, in this case the return value is appropriate for received API result code. The required mocked data source functions are used at the beginning of each test function.

2. View models

The structure of the view model tests differs slightly from the repository tests. Firstly, the view model tests require some rules which provide flexibility to the code, reduce boilerplate code, and allow to perform special actions, such as starting an activity. The *MainDispatcherRule* had to be created for the purposes of those tests in order to set the main dispatcher to an unconfined one, meaning that no particular thread is used when executing the test code. This is required in order to guarantee that the dispatcher we pass into the view model class will be the only dispatcher used, as the tests must be executed on the main thread.

Next, the parameters of the view model class must be mocked. Those are mostly repositories, as well as event implementations, and lastly the dispatcher. Those mocked classes and parameters are passed into a new instance of the view model class.

As in the case of repository tests, the functions of the mocked classes must be mocked as well. Such functions return a selected value each time the mocked function is called.

In the test cases, the mocked repository and event functions are used at the beginning to set up an appropriate environment. Then, the created view model class is used to initialize and

call selected functions for each test case. The results from the exposed ui state are gathered, and the test asserts whether they equal the expected values.

UI tests

The UI tests use the same technology and methodology as the unit tests.

UI tests are instrumented tests which check the behavior of the UI. They are run on an emulator. The purpose of UI testing is to verify that the selected screen displays appropriate data for a given state.

To perform the test, a **compose rule** is required, which allows us to set up the content of the application in a test environment. Testing the UI requires setting special tags on the composables in order to identify them on the screen. The tags are set using a Modifier function *testTag*. The values of those tags are used in the test to check the visibility, placement, responsiveness and the overall look.

After setting the content, each test case may perform interactions and assert whether the desired effect is visible on the screen.

User tests

Overview

To assess the effectiveness and user satisfaction of our application, two surveys have been created. They were filled out by a group of 20 participants in different age groups and various technological backgrounds. The goal was to capture potential flaws in application design and operation.

Each participant was assigned specific tasks to perform in the application. The completion times and user satisfaction of selected tasks have been measured with the use of the first survey. The second has questions about the general user experience throughout the entire app. The aim was to achieve at least 80% user satisfaction score from the tasks and general user experience, as it reflects a positive reception of the software.

The examined tasks were:

- the registration process (Task 1)
- adding a new project and dashboard (Task 2)
- adding a new component to the dashboard (Task 3)
- reordering the component list (Task 4)
- inviting a new user into the project (Task 5)

For each task, the participants were asked to complete it, and then asked to grade their experience on the scale 1-5 where 1 is the worst experience and 5 is the best. Additionally, the time to complete the task was measured in seconds.

Results

Fig. 38. The dependency of user satisfaction score based on age of the participant for each task, measured in the tasks survey.

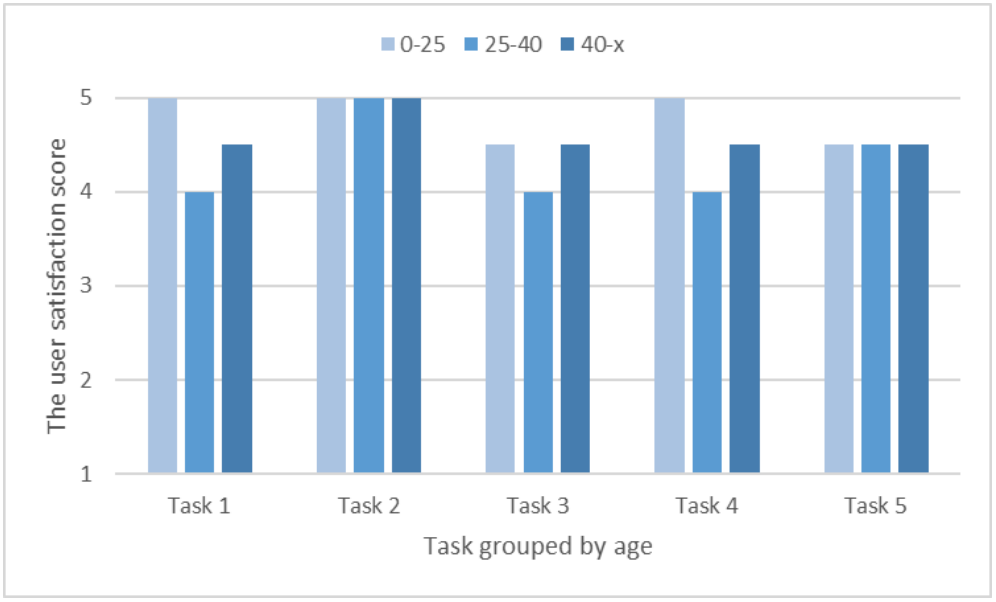


Fig. 39. The dependency of task completion time based on age of the participant for each task, measured in the tasks survey.

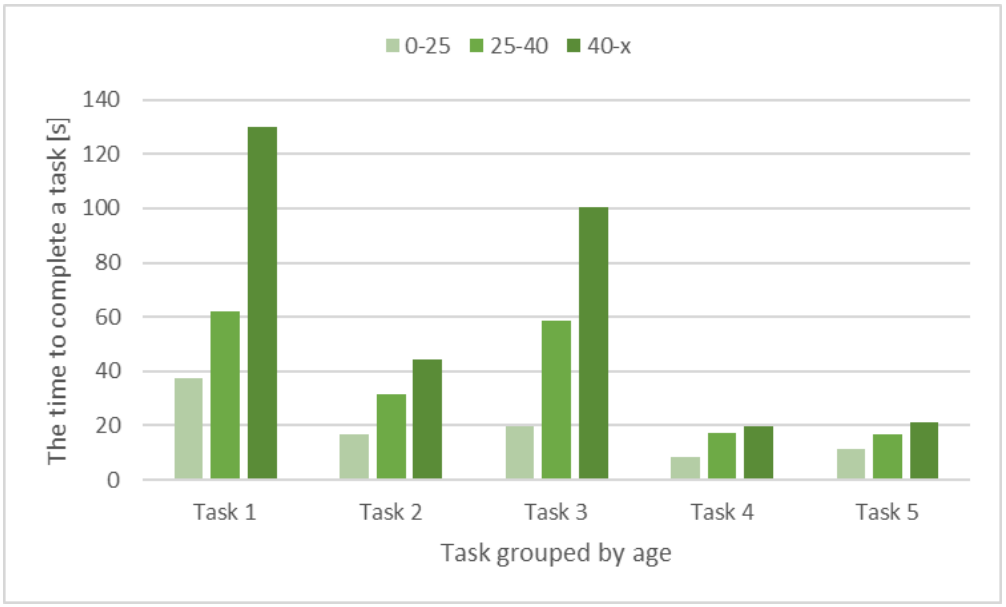
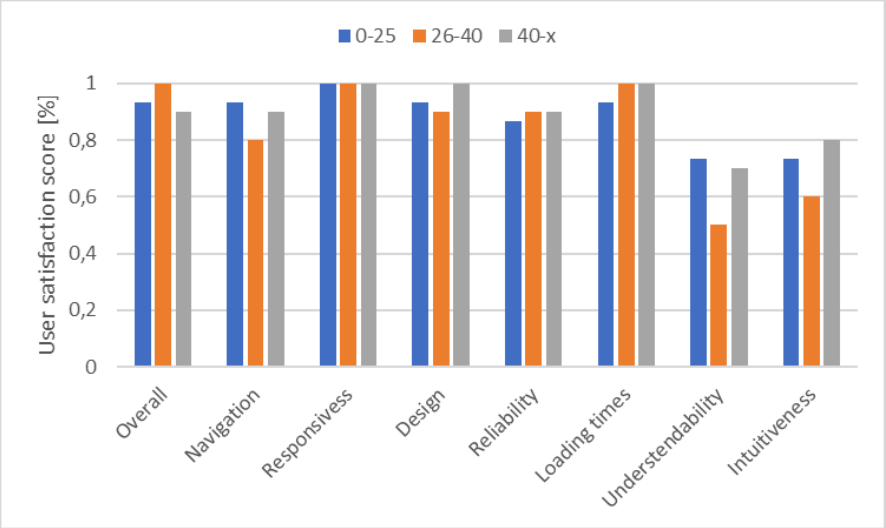


Fig. 40. Table presenting the average user satisfaction score, measured in the tasks survey.

Age group	0-25	26-40	40-x	**Total**
User satisfaction score	0.96	0.86	0.92	**0.91**

Fig. 41. The dependency of user satisfaction score based on the age of the participant, measured in the general survey.



On the graph in fig. 38. it can be seen that for each task, the user satisfaction score was above or equal to 4. Task 2 had the highest scores for each age group, all achieving the maximum score. The middle age group has rated the most tasks the lowest of 4 points.

The graph in fig. 39. shows a clear dependence of the time to complete a task based on the age of the participant. For each task, the oldest participant took the most time to complete it, while the youngest took the least time. Task 1 was the longest for the third age group, as well as the largest time difference of around 70 seconds between the other groups. Task 3 had the largest time difference for the youngest age group, as the time was much lower than for the other groups, by around 40 seconds. The tasks 4 and 5 took the least amount of time overall for all groups.

The last graph in fig. 41 it can be seen that responsiveness and loading times have the highest results, while understandability and intuitiveness have the lowest for all age groups. Responsiveness received a perfect score of 100% from all age groups.

Conclusions

A wide range of tested functionalities enabled us to evaluate application performance in different scenarios. The gathered data shows that each performed task received a very high user satisfaction score with the smallest result being 4 from all age groups. This implies that all of the core functionalities of the application are easily understandable by the user, and they are designed well which makes the experience good. This is additionally held by the low completion times of the task. The longest task is actually the registration, which is a standard process in many apps and it requires a large amount of data from the user, hence the time.

The overall user satisfaction score is 91%, which is over the desired 80% level. The users graded their experience as very good. The lowest score was for the middle-aged group with a score of 86%, while the highest for the youngest group with a score of 96%. Overall, the application has been received very well.

The general surveys show a similar result, as most of the criteria has been graded above 80% user satisfaction score. The participants graded the speed and responsiveness of the application as the best, while understandability and intuitiveness as the worst. This may be because the app requires technical knowledge in order to operate, as it is directed at developers of IoT devices. Users that have never needed similar software may find it confusing as to what the purpose of the application could be. However, this data also shows that our application returns results very quickly, is reliable and well designed, which is why the overall user satisfaction score is very high and exceeds our expectations.

Result demonstration

Overview

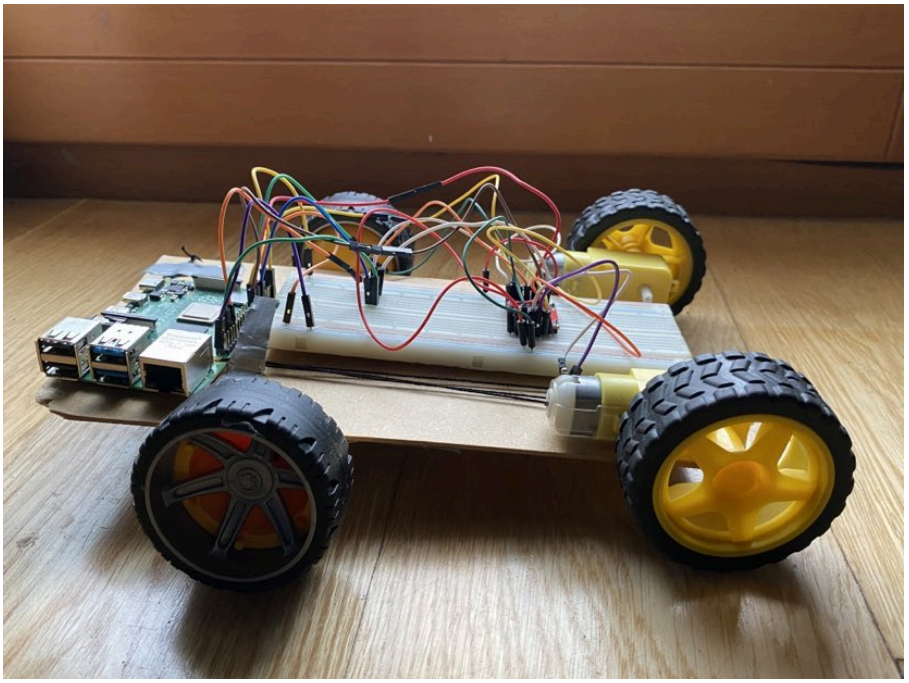
In order to demonstrate the functionality of our application, our team has developed an example IoT device and connected it into the system to communicate with the mobile device.

The IoT device is a driving robot on four wheels, designed to be started and controlled from the mobile application. From the mobile application, we can turn the robot on and off, set the speed of the robot, as well as make it turn left and right.

Construction

The core of the robot is Raspberry Pi 4, which has been programmed to control the movement of the robot. It is built using 2 motors which are connected to a motor controller. The appropriate pins from the Raspberry Pi are connected to the pins of the controller, in order to power the device, and take control of the two separate motors.
Each motor requires two cable connections to the pins of the controller. Those pins connect to the pins of the Raspberry Pi and are used in code to turn the wheels in the right direction. There is an additional pin connection for each motor to the controller which defines the PWM signal, controlling the speed of the device.
The device has a single source of power which is directed to the Raspberry Pi.

Fig. 42. The created robot for demonstration.



Code

The code for the robot required mainly the setup of GPIO pins and of asynchronous processing. There are multiple WebSocket sessions established to the Backend Server. Special endpoints designed for IoT devices are used. On connection, for each session, there is one JSON sent to the Backend Server with the request for Messages. Then, after every Message comes from the server, an appropriate callback is called. These callbacks modify the configuration of the GPIO pins so that the robot behaves in a way determined by the user interaction. There is a separate callback for each session. After the program is finished, a cleanup takes place. The whole implementation required about a 100 lines of code.

Mobile application setup

The mobile application is the crucial part of the demonstration. In order to connect to the robot, we have set up a new project and a dashboard. There, we created 4 new topics: power, speed, left and right. Those are the same topics that are in the code of the device.

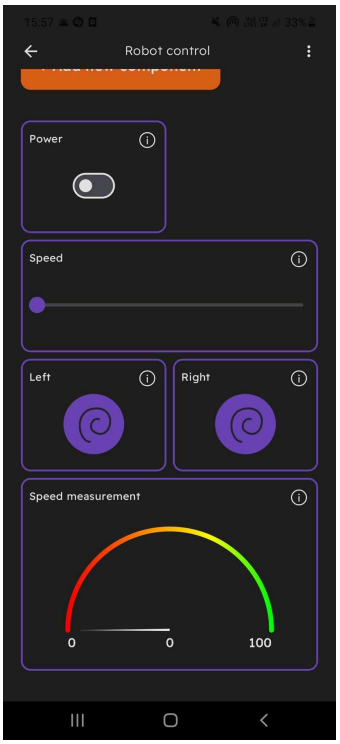
In order to send messages to those topics, we have created 4 components:

- a toggle for the power topic,
- a slider for the speed,
- two release buttons for the left and right.

This completes the required interface needed to create a meaningful connection with the IoT device. Additionally, we have added graph components which monitor the speed of the device.

In order to control the robot, we first have to toggle the power on. Then we can select the speed at which the robot should drive, and at that point the robot starts moving forward immediately.
The additional two buttons make the robot turn left and right accordingly while driving.

Fig. 43. The application screen showing the implemented dashboard with components used to control the robot for demonstration.



Conclusions

Using our system, setting up the control of the robot was very easy. Only around a 100 lines of code needed to be written on the device to enable constant connection with the robot. Creating the interface on the mobile application took around 2 minutes only. Therefore, the process of creating a connection has been very quick and efficient. Recreating this process without our system would require writing much more code in order to effectively send and receive messages, as well as inventing a user interface allowing for such communication. Using our system speeds up this process greatly.

Research and development

The purpose of the System is to create an intuitive and quick to use interface for communication with IoT devices, therefore, the flexibility of the project can be further improved in order to accommodate Users and introduce new functionalities.

Currently, the Mobile App is only available on Android devices. In the future, the number of platforms can be increased to IOS and web applications, to expand the user base. This would also further improve the team collaboration on the app's projects.

The backend application can also be enhanced to allow more network traffic in case of a large number of users. The design of the update propagation could be improved by using RabbitMQ queues to transmit these updates (similar to how Message propagation works right now). This would improve the performance of the process and allow for a larger scale of the usage of the System.

To further develop the team collaboration feature, and enable more Users from non-technical backgrounds to use the Mobile App, a new system for sharing and copying projects could be developed. The idea is to allow the Users to create a working project in the Mobile App, and share, using e.g. a special link, the results to people who will then only use the setup dashboards and components to use with the robot. Those Users will not be required to have any programming experience, and will be able to simply interact with their IoT devices as they wish. This would be a great expansion of the System, as it would target a completely new group of people than currently, and simplify the process of interacting with IoT devices for many Users.

Additionally, such sharing capabilities could be used to transfer the ownership of projects, and allow new teams to collaborate on the work of others.

Lastly, new types of components could be added to the application, such as live camera feed, audio representation, text messages to and from IoT devices, as well as new integrations with various different systems.

10. Summary

The System provides all features listed in the functional requirements, and in some areas it exceeds the assumed scope of the project. The System was thoroughly tested in terms of quality, which proved it to outperform original expectations in usability and performance. The user tests showed that UX design was well thought out and properly implemented.

The project required more work than expected as the implementation appeared to be more complex than originally assumed. The complexity of the implementation resulted in better quality of the System.

To fulfill all the requirements, it was necessary to use many original solutions. Many solutions required knowledge far exceeding the scope of the studies.

USER DOCUMENTATION

1. Introduction

Short guide on how to use the System.

2. Installation

2.1. System requirements

Backend Server

System
Virtualization enabled

Docker
Docker (minimal version 20.10.12) installed and running

Internet access
Necessary for connection to external web services

Mobile Application

System
The minimum Android version supported by the application is API level 26 (Android 8.0 Oreo).
Designed and tested for Android up to version 12 (API level 34).

Internet access
Necessary to authenticate and utilize application functionalities

2.2. Installation procedure

Backend Server

The production version of the Backend Server is running on the server accessible via the Internet. It is receiving requests on the IP address 10.90.50.102, port 80. To send requests, Global Protect needs to be installed on the host, and it needs to be connected to the network 10.90.50.0/24.

It is also possible to run the production, or development version of the Backend Server on any host fulfilling the prerequisites. It is necessary to take these steps:

1. Download the code from this repository: https://github.com/przemek06/iot_mobile. Use the main branch for the development version, and release branch for the

production version. Port for the development version is 8080, and for the production version is 80.
2. Determine your host's IP address.
3. Create a Discord application, and provide it with this callback http://IP:port/anon/discord. Change the variable discord.oauth.url in Spring properties file to contain this defined callback. Save the bot token. Save the IP address and port (necessary for the Mobile Application setup).
4. Create the .env file (dev.env or prd.env). It should be in the following format:

   SPRING_PROFILES_ACTIVE=prd/dev
   SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/postgres
   SPRING_DATASOURCE_USERNAME=postgres
   SPRING_DATASOURCE_PASSWORD=placeholder
   POSTGRES_USER=postgres
   POSTGRES_PASSWORD=placeholder
   DISCORD_TOKEN=placeholder
   LOCAL_APPLICATION_MAIL_USERNAME=placeholder
   LOCAL_APPLICATION_MAIL_PASSWORD=placeholder
   RABBITMQ_DEFAULT_USER=guest
   RABBITMQ_DEFAULT_PASS=placeholder

   SPRING_DATASOURCE and POSTGRES properties should be the same. Replace the "placeholder" with desired values. LOCAL_APPLICATION_MAIL should be the real email account. It needs to have a "third party access" option turned on.
5. Build the application by running a command "docker-compose build" in the root directory.
6. Run the Backend Server by running a command "docker-compose up" in the root directory.

Mobile Application

As the application is not available through the Google App Store, you must download the file containing the application through the Internet.

In order to install files from unknown sources, the option must first be allowed:

1. Go to settings on your device and tap **Apps & Notifications**.
2. Enter the menu using the **three dot**s in the upper right corner.
3. Tap **Special access**.
4. Tap **Install unknown apps**.
5. Tap the browser of your choice.
6. Toggle **Allow from this sourc**e to the **On** position.

With the permission allowed, now you can download the file on your Android device:

1. Open the following link in the browser on your Android device: https://github.com/przemek06/iot_mobile/blob/release/frontend/app/release/app-release.apk
2. Download the file.
3. Open file explorer.
4. Go to the Downloads folder and locate the file.
5. Tap on the file and then press **Install**.
6. If the app asks for any permissions, allow them.

Now the application should be installed on your device.

In order to successfully run the application, you must be connected to the **rynsztok.santos.pwr.edu.pl** network via **Global Protect**.

2.3. Description of the implementation of typical tasks organized according to their types and/or actors

Initially, the User must create a new account in the system. He must fill out all required data in registration, and activate the account. Once logged in, the User should read the explanatory information which he can redirect to from the main page of the application. Then, the User is directed to the project page where he should add a new Project. After opening the project details page, the User should create a new dashboard, and add some components there. The components along with the Topics created for them should resemble the functionalities and connection links that will be used to communicate with the desired IoT device. Having this, the user may set up the IoT device using the created Topics and the project's connection key in order to enable communication. The user may also invite other Users to collaborate on the project.

There are two ways of communication between the IoT device and the System. The device can send Messages to Topic with HTTP protocol, or receive Messages from Topics using the WebSocket protocol. To send or receive data, a developer will need a connection key for a given Project, and a unique name for a chosen Topic. They can both be retrieved from the project details page.

First, a program needs to request an appropriate Topic with this endpoint: */anon/topic/device*. The following JSON has to be sent:

Fig. 44. JSON.template necessary for connections from IoT devices

```javascript
{
	"connectionKey": connectionKeyValue,
	"uniqueName": topicUniqueNameValue
}
```

Using the retrieved object, it is possible to construct a Message in the following format:

Fig. 45. Message JSON

```javascript
{
	"topic": topicObject,
	"message": messageContent,
	"connectionKey": connectionKeyValue,
	"tsSent": ""
}
```

This Message can be sent to the System with the use of */anon/messages/device* endpoint.

To listen for Messages on a particular Topic, a program needs to create a WebSocket connection to an endpoint */messages/device* and send the JSON from fig. 44. after the connection is established.

AGILE METHODOLOGY

1. Introduction

Agile methodology is a flexible approach to project management that emphasizes iterative development, collaboration, and customer feedback. In an IT student project, adopting Agile can bring several benefits [51]:

Iterative Development
Agile breaks the project into smaller increments (sprints) allowing for continuous development and frequent releases of working software. This helps in adapting to changes quickly.

Customer Involvement
Agile involves stakeholders throughout the project, ensuring that their feedback is incorporated into the development process.

Adaptability
Agile allows for flexibility. When there are uncertainties or changes in requirements, Agile's adaptive nature enables developers to respond effectively.

Collaboration
Agile encourages teamwork and collaboration. Roles are often interchangeable, allowing individuals to contribute to various aspects of the project.

Continuous Improvement
Through regular retrospectives, Agile teams reflect on their processes and make improvements, promoting a culture of continuous learning.

2. Adopted practices

Regular meetings
We were conducting regular meetings with our supervisor and teachers to get necessary feedback and improve the quality of our project.

Adaptation
We were reacting to the feedback we received from teachers and problems that we encountered on the way. Thanks to this we could deliver our product on time with high feature coverage. For example we needed to change to WebSocket protocol even though we planned to use RSocket initially.

Roles and Responsibilities
We assigned ourselves a role to better understand the scope of our responsibilities and share tasks.

Sprints
We were working in one week sprints, and delivering smaller tasks at the end of each one. It helped us to break down our work and focus on smaller goals.

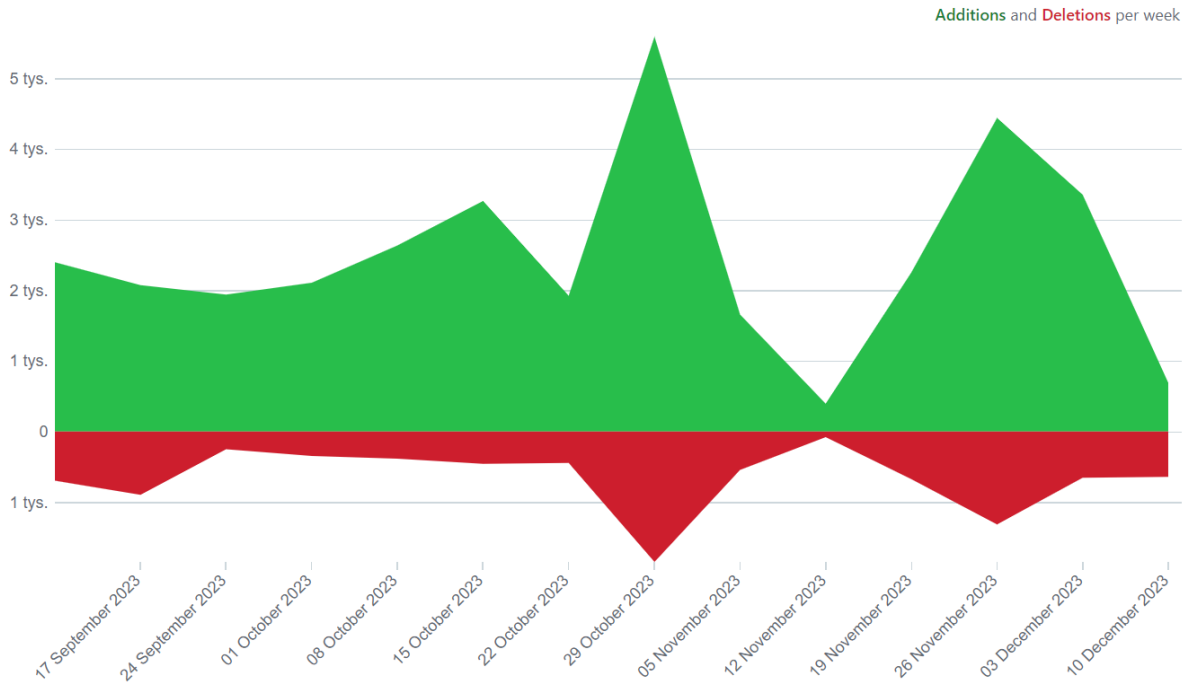
Agile tools
In our project we were working with Jira. From its functionalities we mainly used kanban board for sprint planning and progress measurement, and confluence for preparing documentation and planning further tasks.

2.1. Code repository

Repository url: https://github.com/przemek06/iot_mobile

In order to effectively collaborate on the project, we have created a Github repository, which allows us to write new functionality and features at the same time, and easily combine the results. Additionally, the repository preserves the history of our work, so the state of the project is always secure.

Fig. 46. The code frequency over the history of the project repository.



On the figure above, the code frequency throughout the project timeline is shown. This data resembles the amount of work put into the project, and reflects the work of our group.

2.2. Tasks

In each sprint we assigned ourselves tasks according to the timeline planned at the beginning of the semester. Tasks were relatively small so we could work in an iterative way and be able to have it done at the end of the sprint. Each task contained a description which helped team members understand the scope during reviews.

To monitor our progress we used kanban board with following statuses: **TO DO**, **IN PROGRESS**, **TO CHECK**, **DONE**

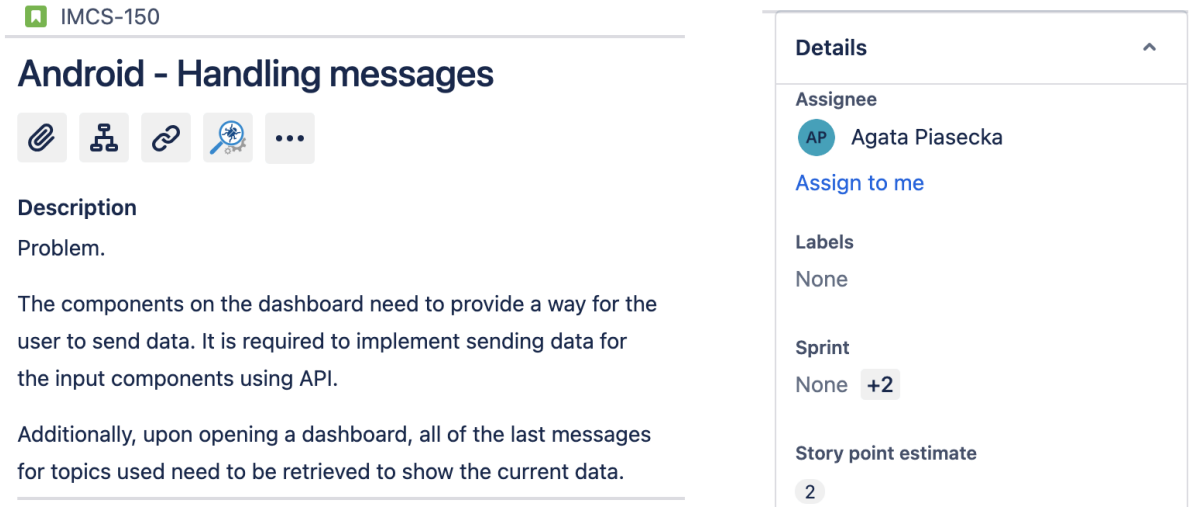
TO DO - task marked as to do was assigned to the current sprint but not yet started. If it had no assignee, it could be assigned to any team member.

IN PROGRESS - this task was already started by the person assigned

CHECK - pull request was prepared and the task was waiting for a review

DONE - task was reviewed and approved by at least one team member, and merged by author

Fig. 47. An example of task definition in Jira.



2.3. Product backlog

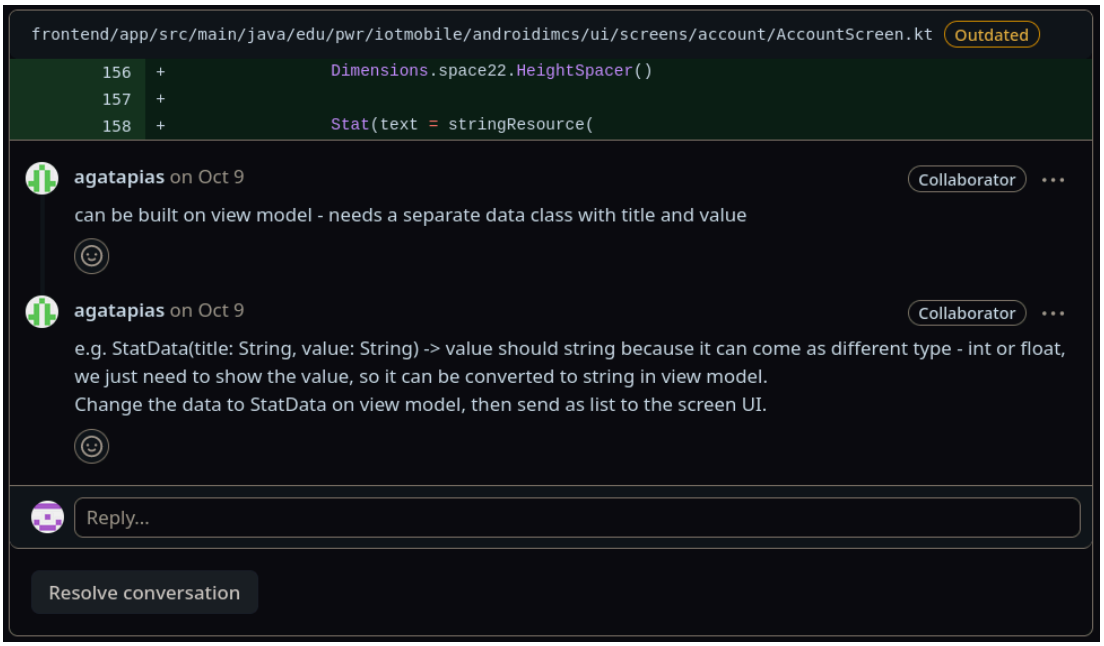
In the product backlog we stored issues prepared based on our functional and non-functional requirements. Most of them were created at the beginning of implementation, but as we were working in agile methodology we were adding new ones if it was necessary, some of them were also modified or deleted.

2.4. Code reviews

For better code quality we were performing commit checking by someone other than the author. This step is crucial for avoiding mistakes that can lead to future problems with development and maintenance of our software. Additionally developers can learn from each other proper development strategies and understanding of the application [52].

As we used GitHub as our VCS, we had access to tools like Merge Request Comments, which really helped in team communication for error spotting and error correction. Reviewers could add comments to specific lines explaining their doubts in implementation.

Fig. 48. An example of code review commentary on Github on a Merge Request.



2.5. Estimations

We estimated our tasks using story points. Each story point corresponded to one hour, where the maximum time allocated for one task was 8 hours. Each team member estimated time for his task independently, so we could assess its level of complexity individually. Thanks to this, the estimates were close to reality, and the tasks were broken down to smaller ones if it could not be done during one iteration.

2.6. Reporting

Burnup reports for last two sprints

Following reports show how we planned our work throughout the sprint. We delivered almost all planned tasks on time, which shows that we managed our time in accordance with task numbers and their complexity.

Fig. 49. The burnup report of the 14th Sprint from Jira.

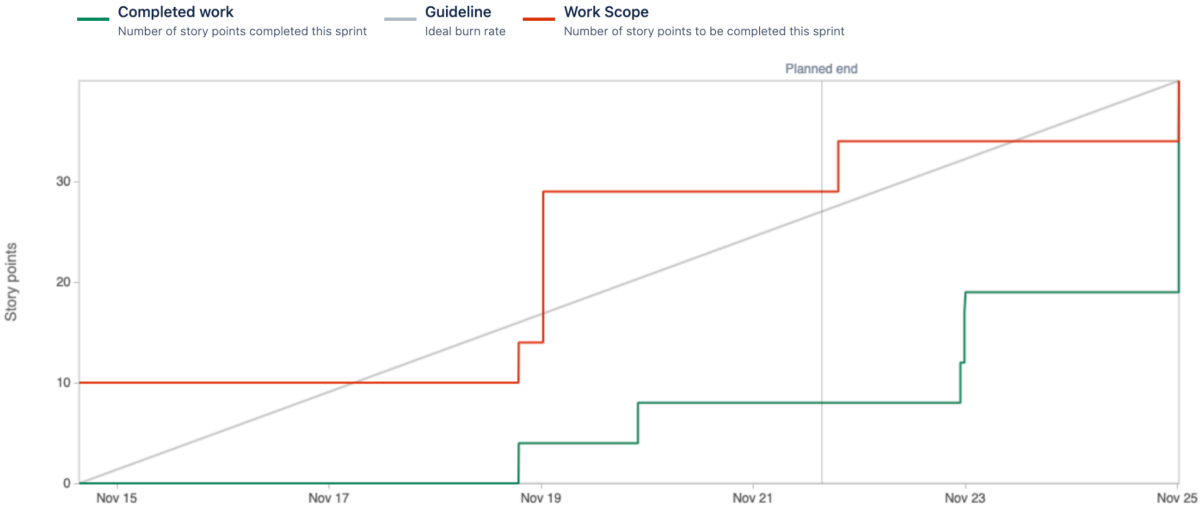
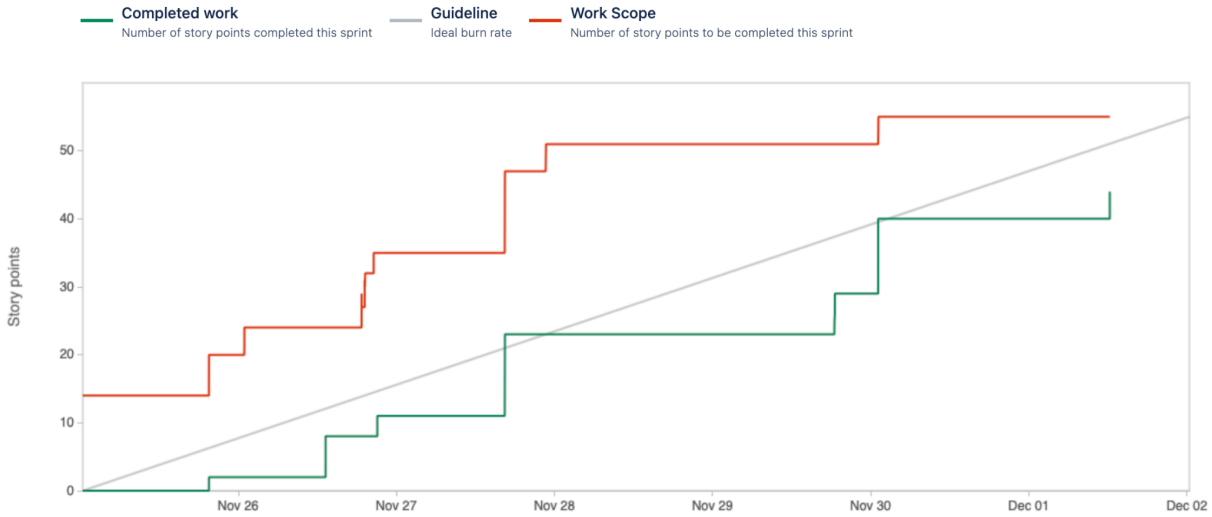


Fig. 50. The burnup report of the 15th Sprint from Jira.



Velocity report

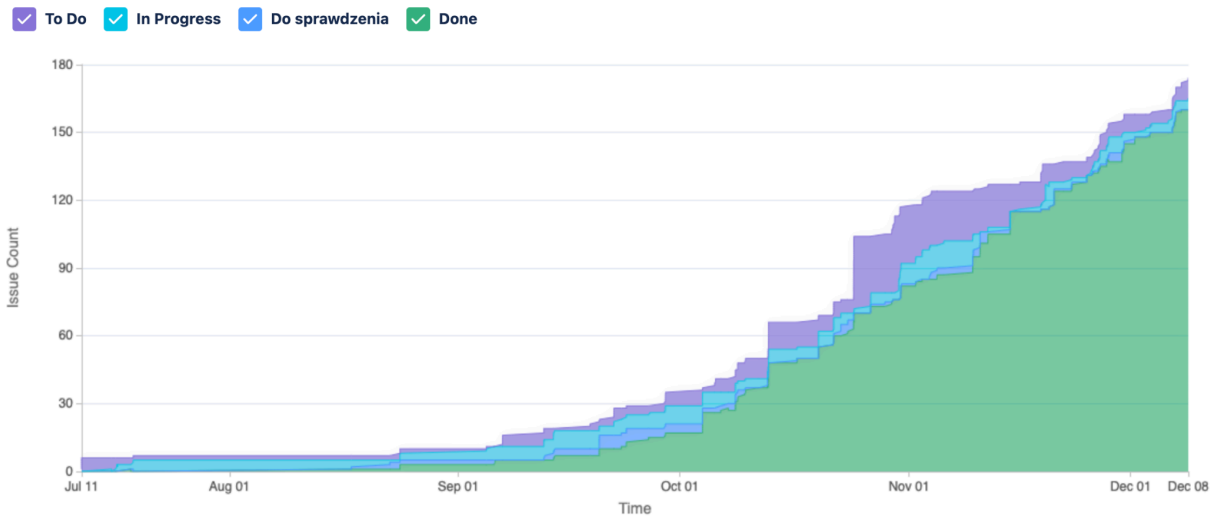
This report shows how much we have done in relation to what we have planned. We can see here that at the beginning of the project (Sprint 10 is the 3rd week of the semester) some tasks were transferred to subsequent sprints, but in Sprint 12 we were able to complete almost all of them. In Sprint 14 we reached all our milestones and there was space for additional tasks in Sprints 14 and 15. Sprint 16 is the last sprint, in which the main goal was finishing up the documentation.

Fig. 51. The velocity report of work throughout the latest sprints.

Cumulative flow diagram

This diagram represents the number of tasks and the change of their statuses during our work.

Fig. 52. The cumulative flow diagram through the project timeline.



CONTRIBUTIONS

1. Planning

Justyna Dul
Contributed in dictionary preparation. Prepared various diagrams including component, use case, entity relationship and database model diagrams. Took part in feature planning and defining functional, nonfunctional requirements and user stories. Was responsible for use case specification.

Szymon Misztal
Prepared list of technologies with their advantages and uses. Prepared stakeholders list. Contributed to user requirements, use case specifications, and documentation. Made a class diagram representing project structure.

Przemysław Świat
Provided the dictionary. Defined the project context. Contributed to functional requirements specification. Created a detailed description of the alternative solutions. Contributed to business goals definition. Refined the non-functional requirements, and use cases.

Agata Piasecka
Created the UI/UX mockup designs for the entire application. Contributed to functional requirements specification. Contributed to business goals definition. Contributed to creating the entity-relationship diagram.

2. Implementation

Justyna Dul
Backend development

1. Defining RabbitMQ and its dependencies in docker compose. Making it easily configurable using application.properties.
2. Creating communication between Backend Server using amqp protocol and other Java libraries.
3. Preparing RabbitMQ administration interface, which was used to manage channels, queues and exchanges and create bindings.
4. Creating interface for sending messages using API.
5. Adding consumers to RabbitMQ topics, which responds dynamically on user request.
6. Forwarding messages stream using WebSocket protocol
7. Creating Slack integration with the use of Slack webhook
8. Creating Telegram integration with the use of Telegram bot, with chat id dynamically retrieved from Telegram API
9. Binding integrations with the Trigger Components
10. Creating unit tests

Szymon Misztal
Main responsibility was:

1. Implementing account management
2. Implementing group management
3. Developing search capabilities
4. Making UI screens and components
5. Adding API integrations for communication with the Backend Server
6. Creation of repositories
7. Adding line graph and speed graph
8. Adding notification system for mobile app with use of WebSockets

Przemysław Świat
Backend development:
1. Containerization of the Backend Server with Docker
2. Implementation of security mechanisms
3. Creation of ~40 HTTP endpoints
4. Synchronization of Dashboard changes, invitations
5. Implementation of WebSocket handlers
6. Mechanisms for operation of Trigger Components with external integrations
7. Integration with Discord, Gmail, and mobile phone notifications
8. Automated tests

Agata Piasecka
Android development:
1. Designed the mobile app architecture
2. Set up the Android project and theme of the application
3. Created the navigation system of the application
4. Implemented the login/register UI & view model logic.
5. Implemented the main screen UI & view model logic.
6. Implemented the projects & project details UI & view model logic.
7. Implemented the dashboard UI & view model logic.
8. Designed and implemented the functionality of list reordering.
9. Implemented the 9 of 12 components UI along with their distinct functionalities.
10. Integrated the system with the native camera application.
11. Developed interfaces to collect data using the WebSocket protocol.
12. Implemented WebSocket clients and UI state changes.
13. Implemented error handling for the entire application.
14. Implemented loading screens for the entire application.
15. Performed unit, UI and user tests.

3. Additional functions

Przemysław Świat
Project management:
- Conducting team meetings
- Synchronization of work of the team
- Making sure adopted team practices are respected

Agata Piasecka
Jira management:

- Defining detailed tasks.
- Creating and finishing sprints, planning work.
- Overseeing project progress.

REFERENCES

[1] J. Chen et al., “Your IoTs Are (Not) Mine: On the Remote Binding between IoT Devices and Users,” scholars.cityu.edu.hk, Jun. 01, 2019. Available at: https://scholars.cityu.edu.hk/en/publications/your-iots-are-not-mine(44e66f6a-0dda-4433-babe-94c67e5c2ec4).html (accessed Dec. 08, 2023).
[2] Ur, B., et al.: Practical trigger-action programming in the smart home. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 803–812. ACM, New York (2014).
[3] H. Assistant, “Documentation,” Home Assistant. [Online]. Available at: https://www.home-assistant.io/docs/. (accessed Dec. 07, 2023).
[4] “Adafruit IO,” io.adafruit.com. [Online]. Available at: https://io.adafruit.com/. (accessed Dec. 07, 2023).
[5] “Adafruit IO API Reference,” io.adafruit.com. [Online]. Available at: https://io.adafruit.com/api/docs/ (accessed Dec. 07, 2023).
[6] IFTTT, “Service API requirements,” IFTTT. [Online]. Available at: https://ifttt.com/docs/api_reference (accessed Dec. 07, 2023).
[7] D. Kirk and S. G. MacDonell, “Investigating a conceptual construct for software context,” Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, May 2014. Available at: https://doi.org/10.1145/2601248.2601263.
[8] S. Benn, R. Abratt, and B. O’Leary “Defining and identifying stakeholders: Views from management and stakeholders” [Online]. Available at: https://www.researchgate.net/publication/305154862_Defining_and_identifying_stakeholders_Views_from_management_and_stakeholders (accessed Dec. 09, 2023)

[9] Leonor Teixeira, Ana Raquel Xambre, Helena Alvelos, Nelson Filipe, Ana Luísa Ramos “Selecting an Open-Source Framework: A practical case based on software development for sensory analysis” [Online]. Available at: https://www.sciencedirect.com/science/article/pii/S1877050915026605/pdf?md5=12e3110cbe00429c1e99fa24d66c76b1&pid=1-s2.0-S1877050915026605-main.pdf (accessed Dec. 09, 2023).
[10] Jetpack Compose documentation [Online]. https://developer.android.com/jetpack/compose/documentation (accessed Dec. 10, 2023).
[11] Spring boot documentation [Online]. https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/ (accessed Dec. 10, 2023).
[12] Kotlin docs [Online]. https://kotlinlang.org/docs/home.html (accessed Dec. 10, 2023).
[13] PostgreSQL [Online]. https://www.postgresql.org/docs/ (accessed Dec. 10, 2023).
[14] Docker [Online]. https://docs.docker.com/ (accessed Dec. 10, 2023).
[15] RabbitMQ [Online]. https://www.rabbitmq.com/documentation.html (accessed Dec. 10, 2023).
[16] Stamelos, Ioannis G and P. Sfetsos, Agile Software Development Quality Assurance. IGI Global, 2007.
[17] C. Larman, Applying UML and Patterns: An introduction to Object-Oriented Analysis and Design and the unified process. 2001. [Online]. Available: http://ci.nii.ac.jp/ncid/BA53443896 (accessed Dec. 13, 2023).
[18] W. Behutiye et al., "Non-functional Requirements Documentation in Agile Software Development: Challenges and Solution Proposal," in Proceedings of the PROFES, 2017, pp. 515-522.
[19] Miles, Russell, and Kim Hamilton. Learning UML 2. 0. O'Reilly Media, Incorporated, 2006.
[20] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 2nd ed. Reading, MA: Addison Wesley, 2003. [E-book] Available: https://people.ece.ubc.ca/matei/EECE417/BASS/index.html (accessed Dec. 13, 2023).
[21] B. Totty, D. Gourley, M. Sayer, A. Aggarwal, and S. R. Reddy, HTTP: The Definitive Guide. 2002. [Online]. Available: http://ci.nii.ac.jp/ncid/BA62153412 (accessed Dec. 13, 2023).
[22] A. Diaconu “The WebSocket Handbook.” [Online]. Available: https://pages.ably.com/hubfs/the-websocket-handbook.pdf (accessed Dec. 13, 2023).
[23] “Getting Started With RSocket: Spring Boot Request-Stream,” Getting Started With RSocket: Spring Boot Request-Stream. [Online]. Available: https://spring.io/blog/2020/03/23/getting-started-with-rsocket-spring-boot-request-stream (accessed Dec. 10, 2023).
[24] T. Collings, “Controller-Service-Repository,” Medium, Aug. 10, 2021. [Online]. Available: https://tom-collings.medium.com/controller-service-repository-16e29a4684e5 (accessed Dec. 13, 2023).
[25] M. Massé, REST API design rulebook [designing consistent RESTful web service interfaces]. Beijing [U.A.] O’reilly, 2012.

[26] Qader, Shko, et al. "Aspect Oriented Programming: Trends and Applications." UKH Journal of Science and Engineering, vol. 6, no. 1, pp. 12-20, Jan. 2022. DOI: 10.25079/ukhjse.v6n1y2022. pp12-20.
[27] “Chapter 6. Aspect Oriented Programming with Spring,” docs.spring.io. https://docs.spring.io/spring-framework/docs/2.5.5/reference/aop.html (accessed Dec. 13, 2023).
[28] Picard, Romain. Hands-On Reactive Programming with Python. Packt Publishing, 2018.
[29] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, Design patterns: elements of reusable Object-Oriented software. 1994. [Online]. Available: http://www.ulb.tu-darmstadt.de/tocs/59840579.pdf (accessed Dec. 13, 2023).
[30] R. Ranjan, “The Mobile App Architecture Guide for 2024,” Insights - Web and Mobile Development Services and Solutions, Nov. 20, 2023. https://www.netsolutions.com/insights/mobile-app-architecture-guide/ (accessed Dec. 11, 2023).
[31] N. Akhtar and S. Ghafoor, “Analysis of architectural patterns for Android Development,” ResearchGate, Jun. 2021, [Online]. Available: https://www.researchgate.net/publication/352021976_Analysis_of_Architectural_Patterns_for_Android_Development (accessed Dec. 11, 2023).
[32] “Guide to Android app modularization”, Android. [Online]. Available at: https://developer.android.com/topic/modularization (accessed Dec. 11, 2023).
[33] M. Bojkić, Đ. PržUlj, M. Stefanović, and S. Ristic, “Usage of Dependency Injection within different frameworks,” ResearchGate, Mar. 2020, [Online]. Available: https://www.researchgate.net/publication/340808780_Usage_of_Dependency_Injection_within_different_frameworks (accessed Dec. 11, 2023).
[34] “Dependency injection in Android”, Android. [Online]. Available at: https://developer.android.com/training/dependency-injection (accessed Dec. 11, 2023).
[35] “Thinking in Compose”, Android. [Online]. Available at: https://developer.android.com/jetpack/compose/mental-model (accessed Dec. 11, 2023).
[36] Robert C. Martin “UML Tutorial: Part 1 -- Class Diagrams” [Online]. Available at: https://www.khoury.northeastern.edu/home/riccardo/courses/csu370-fa07/umlClassDiagrams.pdf (accessed Dec. 09, 2023).
[37] A. Mouat, Using Docker. Sebastopol, Ca: O’reilly, 2016.
[38] “Spring Security :: Spring Security,” docs.spring.io. https://docs.spring.io/spring-security/reference/ (accessed Dec. 10, 2023).
[39] C. Skanda, B. Srivatsa and B. S. Premananda, "Secure Hashing using BCrypt for Cryptographic Applications," 2022 IEEE North Karnataka Subsection Flagship International Conference (NKCon), Vijaypur, India, 2022, pp. 1-5, doi: 10.1109/NKCon56289.2022.10126956.
[40] “Save data in a local database using Room”, Android. [Online]. Available at: https://developer.android.com/training/data-storage/room (access Dec. 11, 2023).
[41] Q. Liu and X. Sun, “Research of Web Real-Time Communication based on Web Socket,” Int’l J. of Communications, Network and System Sciences, vol. 05, no. 12, pp. 797–801, Jan. 2012, doi: 10.4236/ijcns.2012.512083.
[42] M. Vivo, “ViewModel: One-off event antipatterns - Android Developers - Medium,” Medium, Jun. 15, 2022. [Online]. Available: https://medium.com/androiddevelopers/viewmodel-one-off-event-antipatterns-16a1da869b95 (accessed Dec. 11, 2023).
[43] “Scopes | Koin.” https://insert-koin.io/docs/reference/koin-core/scopes/ (accessed Dec. 11, 2023).
[44] “Service”, Android. [Online]. Available at: https://developer.android.com/reference/android/app/Service (accessed Dec. 11, 2023).
[45] Z. Wang and Y. Sun, “How to design the registration and login function of APP,” Journal of Software Engineering and Applications, vol. 11, no. 05, pp. 223–234, Jan. 2018, doi: 10.4236/jsea.2018.115014.
[46] C. Neves, C. Lin, et al., “A Study on the Energy Consumption and Performance of Single-Activity Android Apps”, 7th International Workshop on Green and Sustainable Software (GREENS), 2023, doi: 10.1109/GREENS59328.2023.00008.
[47] “Bottom Navigation Bar in Android”, Android. [Online]. Available at: https://www.geeksforgeeks.org/bottom-navigation-bar-in-android/ (accessed Dec. 11, 2023).
[48] “Design systems in Compose”, Android. [Online]. Available at: https://developer.android.com/jetpack/compose/designsystems (accessed Dec. 11, 2023).
[49] “Choosing A Load Testing Strategy.” [Online]. Available: https://bento.cdn.pbs.org/hostedbentoga/filer_public/smoke/load_testing.pdf (accessed Nov. 13, 2023).
[50] “Functional Testing.” [Online]. Available: https://www-users.cse.umn.edu/~heimdahl/csci5802-spring02/readings/book-ch13-functional.pdf (accessed: Dec. 12, 2023).

[51] Martin, Robert. Clean Agile: Back to Basics. Pearson, 2019.
[52] Larry Conklin and Gary Robinson “CODE REVIEW GUIDE 2.0” [Online]. Available at: https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf (accessed Dec. 09, 2023).