HAW HAMBURG

BACHELOR THESIS
Andrea Minguez Angulo

Design and implementation of a Flutter-based mobile App for SCRUM project management

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Andrea Minguez Angulo

Design and implementation of a Flutter-based mobile App for SCRUM project management

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of European Computer Science*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Stefan Sarstedt
Supervisor: Prof. Dr. Ulrike Steffens

Submitted on: 03. August 2023

Andrea Minguez Angulo

Title of Thesis

DESIGN AND IMPLEMENTATION OF A FLUTTER-BASED MOBILE APP FOR SCRUM PROJECT MANAGEMENT

Keywords

SCRUM, Task Management, Scrum Master, Product Owner, Developer, Sprint, Mobile App, User Interface (UI), Flutter, Widget, Firebase.

Abstract

This thesis focuses on the design and implementation of Scrumer, a mobile application aimed at supporting agile project management using the Scrum framework. The objective of this research is to develop an intuitive and user-friendly mobile app that facilitates efficient collaboration and task management within Scrum teams, while also providing an introduction to the Scrum framework for users who utilize Scrumer for personal use.

The study begins with an analysis of existing project management apps, both related and unrelated to Scrum. Based on this analysis, Scrumer is developed, incorporating Scrum principles while offering flexibility for regular users and those involved in Scrum teams.

The functional requirements of Scrumer are defined, including user registration and login, project creation and management, task creation and collaboration, and sprint planning. Non-functional requirements, such as cross-platform compatibility, responsive design, and intuitive user interfaces, are also considered.

The development phase includes the analysis, design implementation and evaluation of the app. The analysis process defines the class architecture and identifies the use cases. This is followed by the design process, where the system architecture, user interface, and logo of Scrumer are created. The implementation of the system architecture and user interface design successfully meets the established requirements, resulting in a functional task management app.

The results of this research demonstrate that Scrumer effectively addresses the needs of Scrum teams by providing a comprehensive set of features for efficient task management, collaboration, and progress tracking. The user-friendly interface and intuitive design of the app contribute to improved productivity and user satisfaction within Scrum teams, as well as for new users adopting a more flexible Scrum framework.

Table of Contents

List of Figures

List of Tables

Part I

Prologue

1 Introduction

To develop our app effectively, we need to establish its underlying motivation and define clear goals for the project. Additionally, analyzing the potential impact of the app is essential. Finally, we need to outline the structure of the development process and of the document by establishing a methodology.

1.1 Motivation

1.1.1 Why make an App?

Since I began studying software engineering, I have always had a strong desire to develop a project from scratch to showcase the skills I have learned. My interest in the world of app development has consistently grown, along with its potential.

1.1.2 Why make an App for SCRUM Project Management?

As I learned about SCRUM methodologies, I started applying them not only for teamwork at university, but also for personal projects and daily tasks. Implementing time management using sprints significantly improved my task management efficiency.

The idea of having this methodology available on our mobile phones interested me. I realized that it could benefit not only professionals working within the SCRUM framework but also individuals across various domains, even for personal projects unrelated to work.

Therefore, the motivation behind developing this app stems from the need to provide users with an efficient tool to manage their tasks within the SCRUM project management framework. The key motivations are as follows:

- Make task Management productive: By using the SCRUM methodologies such as user roles and sprints, the app will provide a different way of managing tasks, making it more organized and time-focused.
- Make collaboration a key element: By allowing multiple users to work on the same project, the app will facilitate teamwork. Features such as real-time updates and a comment section for each task will enable seamless collaboration and communication among team members.
- Make an innovated Task Management system for all Users. The app will feature an easy-to-use interface, making it accessible to both users within the SCRUM framework and casual users. It aims to provide an innovative and user-friendly task management experience, regardless of users' familiarity with SCRUM.

1.1.3 Why make a Mobile App?

The motivation behind developing a mobile app stems from the need for a task management tool that is always available and easily accessible to users.

The advantage of having a mobile app instead of relying solely on a web-based platform is the ease and speed with which users can access and update their tasks. For instance, users could quickly check the progress of tasks within a sprint by simply opening the app on their phones, eliminating the need to access a website and log in to perform a swift check.

By developing a mobile app, we can provide a seamless and user-friendly experience, enabling individuals to effortlessly manage their tasks, track progress, and stay organized; minimizing the time and effort required to manage their tasks.

This accessibility aligns perfectly with the fast-paced modern life, allowing users to efficiently handle their workload or personal projects anytime, anywhere.

Furthermore, by utilizing the features of mobile devices, such as the camera, our app can offer additional functionalities that enhance the user experience. For instance, integrating QR code reading or photo-taking capabilities.

Therefore, providing a mobile app offers the best solution for users of our app.

1.2 Goals

In this thesis, two main goals are crucial for the project's success:

- Successful App Design and Implementation: The primary goal is to successfully design and implement the app, ensuring its functionality, usability and reliability.
- Detailed Documentation: In parallel with the app development, a comprehensive and professional documentation process will be followed. The goal is to provide a detailed insight into the development process, methodologies used, decisions made and the final conclusions.

To achieve the goal of successful app design and implementation, the following specific goals have been identified:

- Requirement analysis: conduct a profound analysis of the app's requirements considering all target users.
- Design: conduct a design for the App regarding the user interface, logo and overall architecture.

  The UI design should be intuitive, visually appealing, and aligned with the guidelines for a positive user experience.

  The logo design should be representative of the app's purpose and brand identity.

  The architecture design should consider factors such as the different users on the App, future new features, performance and maintainability.
- Implementation:develop the app based on the established design and requirements. This process involves writing the code, establishing the database, utilizing the appropriate packages and implementing the App's functionalities.
- Evaluation: Evaluation of the App, to observe if all of the requirements are met.
- Final result: working App with an intuitive user interface.

1.3 Impact

The expected impact of the App, is solely aimed at the User. The goal is to answer to the question: "How does the use of the App affect the productivity of Task Management within a Scrum cycle?"with a positive response, making an actual change in users daily life.

We want to change the way the tasks are being managed by making it more productive, allowing the user to instantly check the SCRUM cycle state and make instant updates, which other users on the project can also instantly see.

Moreover, within the SCRUM cycle, we want to apply customized features for specific roles. For instance, special attention is given to the Scrum Master role to simplify his/her responsibilities and make project management easier.

Finally, we want to pay special attention to the type of user that is not on a SCRUM project, and it is first introduced to the methodology on this App. We want to have an impact (with the help of the SCRUM framework) on the way he/she structures his/her tasks and work.

1.4 Development Methology

For the development methodology, we utilize a hybrid approach to project management, combining Waterfall and Scrum methodologies. This approach is often referred to as "Water-Scrum-Fall."

In Water-Scrum-Fall, the overall project follows a Waterfall methodology, following a sequential and linear progression of phases. The phases followed are:

1. Requirements Gathering
2. Analysis
3. Design
4. Implementation
5. Evaluation

In each phase, SCRUM principles and practices are applied to allow for flexibility and iterative development. Specifically, during the implementation phase, we obtain the first Minimum Viable Product (MVP) of our App. From there, we iteratively add the necessary features to meet the established requirements and analysis.

This approach enabled us to respond to changes and incorporate feedback throughout the development process, resulting in an App that aligns with user needs and expectations.

1.5 Document Structure

The structure of the document is based on the development methodology.

Having done the introduction, the context in which the App is going to be develop is established, including a short explanation of SCRUM, an overview of Scrumer (our App) and a comparison of other similar Apps.

Once the context has been analysed, the next step is the development.

In the development section, first, the requirements for the app are gathered. Secondly, an analysis is done, including the creation of a use case and class model. Then, the design of the architecture, User Interface and Logo is established.

Then, the implementation process is structure, explaining, first, the build environment and the source code. Then, different systems for managing data from the database and for responsive implementation are implemented. Finally, each page and the launching logo is implemented.

Finally, a evaluation of the requirements takes place, though unity tests, an analysis to observe if all requirements are met, and acceptance testing.

Having done the development of the app, the conclusion takes place, where an analysis of the goals achieved, of the lessons learned and of the future work takes place.

2 Context

Before we begin developing the app, it is important to analyze the current context and understand what we are aiming to build. Therefore, we begin with a definition of SCRUM and how its principles shape the features of our app. We also explore the concept of "Large Scaled Scrumänd evaluate the possibility of building the App based on this approach.

Secondly, we provide a brief overview of how our app will work adapting the SCRUM Rules and the underlying idea behind it.

Finally, we analyze other apps in the market to compare them with our idea and identify what new elements we can bring to the table.

2.1 SCRUM

SCRUM is a framework for the Agile development of projects.[18]

In the book *SCRUM Master* written by Marta Palacio [32], the fundamentals of SCRUM are addressed. It states that SCRUM is a framework for Agile project development. Agile consists of principles and practices for project development, based on four main characteristics:

- Individuals and iteraction over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contact negotiations.
- Responding to change over following plan.

Within SCRUM, projects are developed using a series of work cycles called Sprints.

The goal of each sprint is to deliver a functional product increment. The development process is iterative, meaning that the product is continuously improved with each sprint.

The term SSCRUMïs derived from rugby, where it describes a group of players attempting to gain possession of the ball. This analogy relates to a team of individuals working together to achieve the common goal of a project.

Within the SCRUM Rules, there are three different components or categories: Artifacts, Events and Roles.

Artifacts

Artifacts represent the work or information that is used and produced during the development process. The three primary artifacts in SCRUM are:

- Product Backlog: List of tasks that need to be achieved in the project. It evolves with the client's point of view.
- Sprint Backlog: a subset of the Product Backlog that contains the tasks selected for a specific sprint. It defines the work that the development team will focus on during the sprint.
- Increment: The result of each Sprint. It represents the work that has been accomplished and is potentially shippable to the customer.

Events

Events are time-boxed activities or meetings that occur during the SCRUM process. The five primary events in SCRUM are:

- Sprint planning: Meeting where the goal and tasks of the Sprint are set.
- Sprint: Periods of time in which the work is divided.
- Daily Scrum: Small meeting of 15 minutes to check the progress and plan the next tasks.

- Sprint Review: Meeting to analyze the increment generated. The modification of the Product Backlog takes place if needed.
- Retrospective: Meeting to review of what to improve regarding the SCRUM Rules and how to improve it for the future Sprint.

Roles

Roles define the responsibilities of the individuals involved in the SCRUM process. The three primary roles in SCRUM are:

- Product Owner: Member of the Scrum Team. He/She represents the client's decision. His/Her responsibility is the Product Value. He/She clarifies the product's vision and goal. He/She is responsible for managing the product backlog.
- Developers: Member of the Scrum Team. They are a self-organizing group (3-9 people) responsible for delivering the product increment at the end of each sprint. They collaborate to plan, develop and test the product features. Their responsibility is the Increment. They take part in all of the events.
- Scrum Master: Member of the Scrum Team. He/She coaches the rest of the Scrum Team and moderates the discussions. His/Her responsibilities are the SCRUM Rules.

Cycle

In the SCRUM Cycle, all of the Scrum components are involved in the iterative development process. This is the SCRUM cycle:

1. Product Backlog: The Product Owner establishes the Product Backlog.
2. Sprint Planning: During this meeting, the Product Owner and the Development Team select items from the Product Backlog and create the Sprint Backlog. The Scrum Master moderates the meeting.

3. Sprint: The team collaborates, codes, tests and integrates their work to create the increment.

   Daily Scrum: Throughout the Sprint, the Scrum Team holds daily stand-up meetings.

4. Sprint Review: At the end of the Sprint, the Scrum team analyses the increment.

5. Retrospective: Following the Sprint Review, the Scrum Team holds a Sprint Retrospective meeting.

Once the Retrospective concludes, a new Sprint begins, and the cycle repeats itself.

2.1.1 Large Scaled SCRUM

Large Scaled SCRUM (LeSS) relies on the idea of Scaling Agile, and it is utilized when a whole company follows Scrum.

Agile at Scale is the practice of applying agile principles and processes to larger and more complex projects.[30] The goal is to maintain a consistent level of agility and effectiveness across the entire organization. It may involve multiple teams, projects and tools.

One of the possible approaches to Agile at Scale is Large Scaled SCRUM.[34]

Large Scaled SCRUM is a scaled agile framework that guides companies in adopting and applying agile at scale. It seeks to apply the "principles, purpose, elements, and elegance of Scrum in a large-scale context, as simply as possible."[13]

Its structure is based on Feature Teams, which are in charge of the development work. They are self-managing; stable and long-lived; and cross-functional.

What changes in this approach compared to normal SCRUM is that each of this Featured Teams is a group of developers and there can be one Scrum Master for more than one team. Also, the product Owner is the same for all feature Teams.

Furthermore, there are ambassadors representing each team, as well as a Head of the Product Group, who meet in special meetings for an overall retrospective.

2.2 Scrumer: our App adapted to the Scrum Rules

With Scrumer, our app, we aim to make SCRUM accessible to anyone through their mobile devices. Our goal is to create a task management app based on SCRUM principles, while also allowing flexibility for those who do not strictly adhere to SCRUM rules. Additionally, we want the app to cater to both individual Scrum teams and larger organizations using scaled approaches like LeSS (Large-Scale Scrum).

Regarding the three key components of SCRUM, this is how we could adapt them to or app Scrumer:

Artifacts

- Product Backlog Section: A section where the Product Owner can add tasks, allowing for the prioritization and management of product requirements.
- Sprint Section: Displays the Sprint Backlog, including the tasks assigned to the current sprint and the remaining time. It provides a quick overview of the sprint's progress and allows for efficient tracking.
- Increment: Shows the tasks completed during the sprint, providing visibility into the incremental development of the product. We will display the increment during and after the sprint for a comprehensive view of the progress.

Events

- Meeting Tool: The app can be used during meetings to add tasks to the sprint and set the sprint duration.
- Comments: We also want to empower users to collaborate with the Scrum Team outside of the established meetings. Therefore, users will have the ability to make comments on tasks, promoting collaboration and communication among team members.

To achieve to empower users to adjust values outside of the established meetings, making Scrum flexible, we need features that allow users to: edit tasks titles, sprint duration, add

and remove tasks from the sprint, remove tasks from the product backlog, edit the project title and description and even be able to abandon or delete a project.

Regarding user types, we aim to make the app accessible to users who are not part of a Scrum team and strictly follow Scrum rules. For these users, all the mentioned features will be available.

However, for users within a Scrum team adhering to Scrum rules, different features will be assigned based on their roles.

Roles

In terms of team structure, the Scrum Master will have the responsibility of creating a project. Later on, a Product Owner and multiple developers can join the project. Taking inspiration from the LeSS approach, we will allow the Scrum Master to create other projects, enabling them to be part of multiple projects simultaneously.

Also, every user can be working in more than one project, depending on the type: the normal user (not in a Scrum Team) can create and join projects and the Scrum users, can create project if they are Scrum Masters or join them, if they are Product Owners or developers.

By considering these aspects, Scrumer aims to adapt to both individual Scrum teams and organizations using scaled agile approaches. For the individual Scrum teams, the Scrum Master is only working on one project, so he/she would just create one. And for the Scrum Teams on scaled agile organizations, he/she would be able to manage several teams just by switching between projects.

The app's flexibility and adaptability will empower users to effectively manage their tasks and projects while following SCRUM principles or adapting to their unique needs.

2.3 State of the Art

Before developing our App, it is crucial to identify existing apps that offer similar features. An analysis of these Apps provides an overview of the market and serves as a benchmark for the uniqueness and competitive advantage of your app, Scrumer.

The goal of this analysis is to gain insights into the strengths, weaknesses and innovative features of other Apps, allowing us to position Scrumer effectively.

Considering the existing offerings in the market, the apps that stand out as competitors to our app, Scrumer are *Scrum App*, *Vivify Scrum* and *Toist*.

2.3.1 Scrum App

It is an App available for both Android and IOS devices.

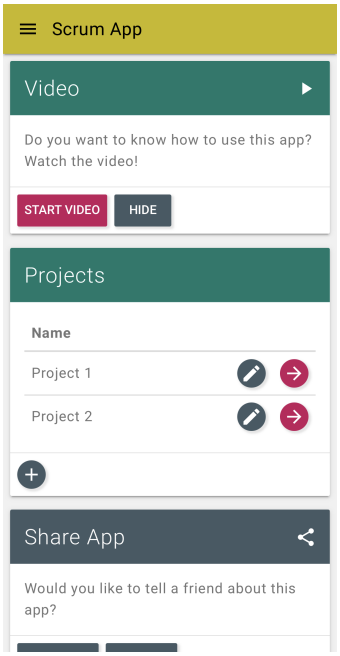
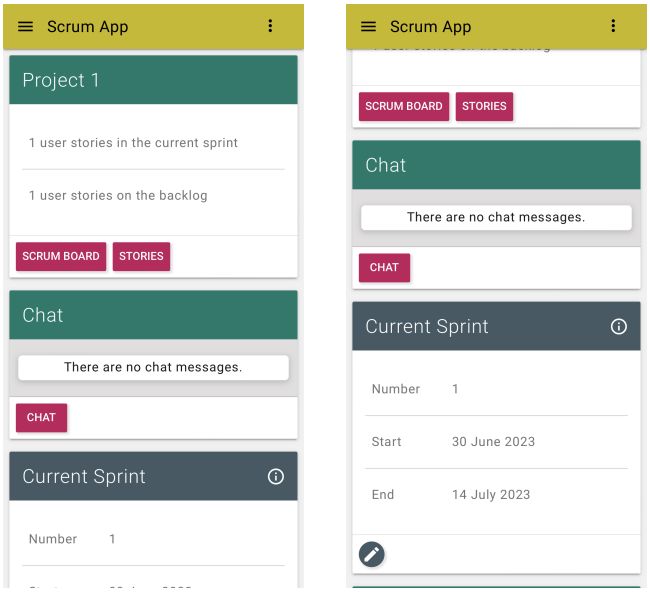


Figure 2.1: Scrum App. Interface with all of the projects

It is based on the Scrum framework. It provides a simple interface, where the user can initially create a new project. We can identify two actions: the one with a tutorial video and the list of all the projects and another section accessed by selecting each project.

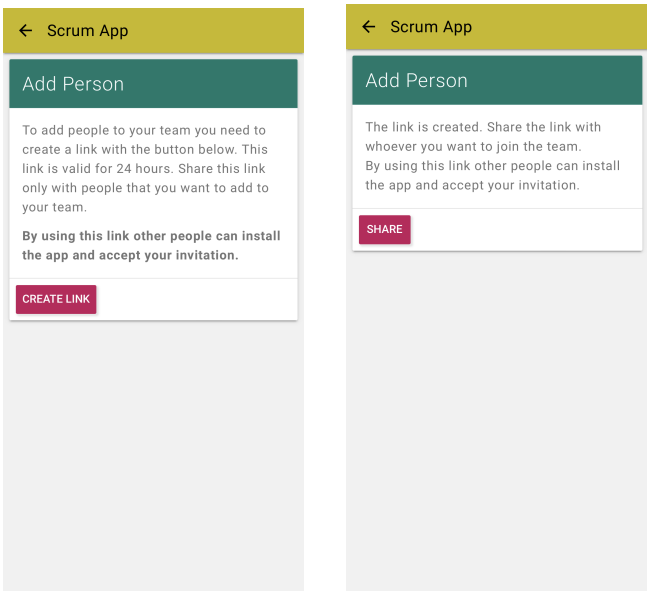
In the section for each project, we can also see different blocks including a task block for adding tasks, a chat, a block for the beginning and ending of the sprint, which can be modified and a block to see the members of the project.

In order to add people to the team, the user can create a link to the project and later, share it.



(a) Tasks and Chat (b) Current Sprint

Figure 2.2: Scrum App. Interface with information of each project



(a) Generate Link (b) Share Link

Figure 2.3: Scrum App. Interface to share a project

We can see that, in this App, they allow the user to make modifications on the project title, on the tasks and also, on the finishing date of the current sprint.

2.3.2 Vivify Scrum

It is an App available for only Android devices.[20]

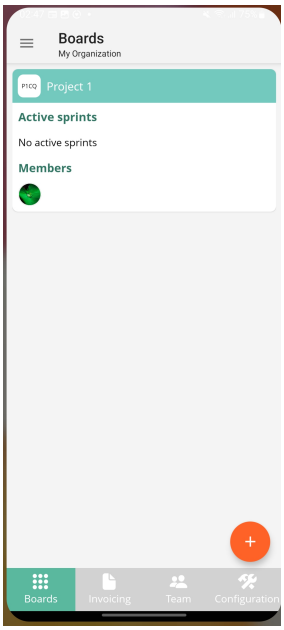


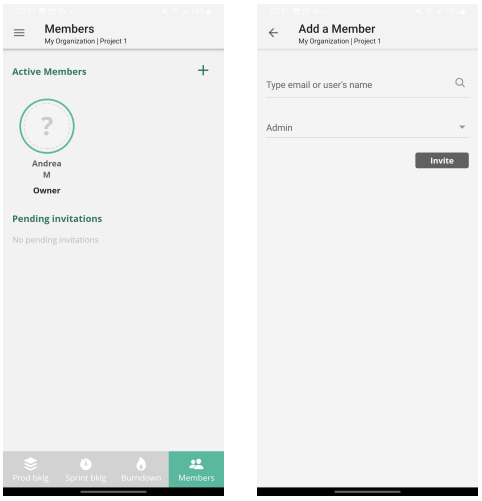
Figure 2.4: Vivify Scrum. Interface with all of the projects

This app works with organizations which can have different Boards (which we could call Projects). We first have a navigation bar with four sections: Boards, where we can create projects; Invoicing, where we see the invoices of the organization; Team, where we see the members of the team; and configuration.

We can access each Board and for each Board, we have a navigation Bar with the product Backlog, the Sprint Backlog, the Burn down and the Members.

To add members to a Board, we can use their email or their user's name, and we can establish their role depending on the permission we give: Admin, Read, Read and Comment.

For each Task, there are many feature. The user can establish where to have the Task (Product Backlog or Sprint), its points of difficulty, value, priority, labels, assign it to performers, reviewers, etc. It does also have a comment section.



(a) Active Members (b) Add a member

Figure 2.5: Vivify Scrum. Interfaces to add a member to a project

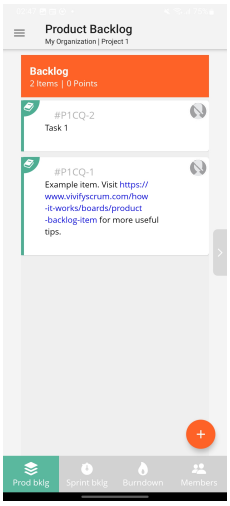


Figure 2.6: Vivify Scrum. Interface to display Product Backlog

2.3.3 Todoist

It is an App available for both Android and IOS devices.[19]

It is not based on the Scrum framework. It provides a simple interface, where the user has a global inbox with all of the tasks. Each task can have a Due Date, a priority and a Label. So, each Label could correspond to each project.

Figure 2.7: Todoist. Interface with all of the tasks

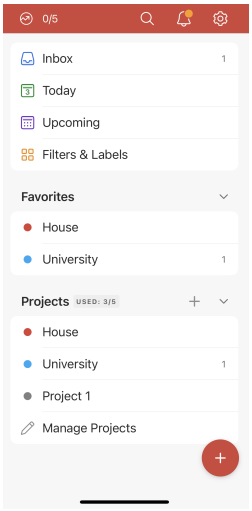


Figure 2.8: Todoist. Interface with the main menu

It has two section, the initial section, where the tasks in the inbox are displayed and a menu, where the user can see the different projects and access the tasks filtering by date, label, project, etc.

The user can also modify the tasks and projects. When a user marks a task as done, the task disappears from the inbox.

2.3.4 Comparison

By conducting a feature comparison among the three apps, we can draw insightful conclusions and ultimately, determine the competitive advantages offered by our app, Scrumer.

Table 2.1: State of the Art Comparison: *Scrum App*, *Vivify Scrum* and *Toist*.

Features	**Scrum App**	**Vivify App**	**Todoist App**
Based on SCRUM	51	51	55
Collaboration on projects	51	51	51
Simplified tasks	51	55	51
Comments on each task	55	51	55
Task prioritization	55	51	55
Task assignment	55	51	55
Deadline management	51	51	51
Sprint Progress tracking	51	51	55
Mobile app availability (Android)	51	51	51
Mobile app availability (iOS)	51	55	51
Scrum roles features	55	55	55

Based on the comparison, we can make a final conclusion on the direction we want Scrumer to take.

While Scrum App and Vivify Scrum are based on SCRUM, neither of them provides Scrum roles.

Scrum App offers a different approach with only 2 sections and a scrolling view for all project information. The idea of clearly separating each project information is an interesting design feature.

Vivify Scrum provides a wide variety of features and targets big organizations, offering a comprehensive app to communicate across all boards in the entire company. However, this is not the intended focus of our app, Scrumer, as we want it to be used exclusively within each Scrum Team, not throughout the entire company. Nevertheless, the design using a navigation bar helps users switch between different sections quickly, which would aid our goal of enabling users to check tasks as efficiently as possible.

Lastly, Todoist is the most widely used app on the market for task management as it has a simplified and clear design for displaying tasks, and the feature of marking a task as done is the most intuitive one.

This analysis demonstrates the opportunity to develop an app like Scrumer that combines the core principles of SCRUM, provides flexibility for users, and supports both regular users utilizing it for personal purposes and users on Scrum teams. By addressing these gaps in the market, Scrumer can offer a unique value proposition to its users.

Part II

Development

3 Requirements

The first step on the development process is the Requirement gathering. Firstly, we will analyze the functional requirements and secondly, the non functional requirements.

3.1 Functional Requirements

These are the functional requirements of the App Scrumer, based on the previous description of the App:

Regarding users:

- FR01: A user has to be able to register on the App with his/her name, surname and email, having available two different registration, depending on the user type:
  - Normal User: the user is not on a Scrum Team, therefore, he/she does not have a Scrum Role. He/She is using the App for personal work.
  - Scrum Team User: the user is on a Scrum Team. He/she has to be able to choose on the registration process, his/her Role.

Table 3.1: User Types Requirement

User Types
Normal User
Scrum Master
Product Owner
Developer

- FR02: A user has to be able to login using the email and password used on the registration.
- FR03: A user has to be able to register and login using a google account.

- FR04: A user has to be able to reset his/her password
- FR05: A user can have multiple projects.
- FR06: A user has the fields Name, Surname and Photo which can be modified.

Regarding projects:

- FR07: A project has the fields Title, Description and Photo which can be modified.
- FR08: A project has tasks and a current Sprint which can also be modified.
- FR09: A normal user can create Projects and Join Projects created by other normal users.
- FR10: A normal user can abandon a project, and when there's no one else on the project, delete the project.
- FR11: A scrum Master can create Projects.
- FR12: A product owner and a developer can join projects.
- FR13: The method to join projects needs to be the following: A QR for the project need to be generated and displayed to be read by another user who uses it to join the project.
- FR14: A project needs to have a mode field, which is "work"when the project has been created by a Scrum Master and "personal"when the creator was a normal user.
- FR15: A "work"project needs to have one Scrum Master and it can have zero or one Product Owner. There can be multiple developers.
- FR16: A product Owner and a developer can abandon a project.
- FR17: A scrum Master can delete a project.

Regarding the current sprint of a project:

- FR18: A sprint has the fields Title, Starting Date and Finish Date, which can be modified.
- FR19: A normal user can create a Sprint and edit its duration.
- FR20: A scrum Master can create a Sprint and edit its duration.

- FR21: All users need to see the progress of the sprint, meaning the tasks completed.

Regarding the tasks of a project:

- FR22: A task has the field Title which can be modified.
- FR23: a task can be created by a normal or a product owner user.
- FR24: A task can be added to or removed from the sprint and deleted from the project.
- FR25: A task has a comment section, which acts as a chat for users on the project to collaborate.

3.2 Non functional Requirements

These are the non-functional Requirements of Scrumer:

- NFR01: The App needs to be compatible with both Android and IOS Operating Systems.
- NFR02: Every screen or component needs to be charged in least that 5 seconds with the aim of not loosing user's attention.
- NFR03: The error messages have to be clear and easy to understand by the user.
- NFR04: The design of the App needs to have a uniform theme, as well as intuitive user interfaces,
- NFR05: Responsive design: the app needs to be adjustable to different screen sizes.

4 Analysis

This chapter covers the analysis of the information system to be developed, making use of the UML modeling language.

4.1 Use Case Model

Based on the functional requirements, we can build the Use case Model.

4.2 Class Model

Based on the App description and the requirements, we can build the UML Class Model.

Analyzing the relations, we can see that all users have from 0 to multiple projects and that a project has at least one normal User when is in "personal"mode; and on "work"mode, it has one Scrum Master and it can have one product Owner and multiple developers.

A Project has an array *sprints*, which can be empty or contain multiple sprints; as well as a *tasks* array which can also be empty or contain multiple tasks.

Finally, a Task can have multiple comments.

To save the passwords of the users for the authentication use cases, a database is used with the emails and passwords of all users.

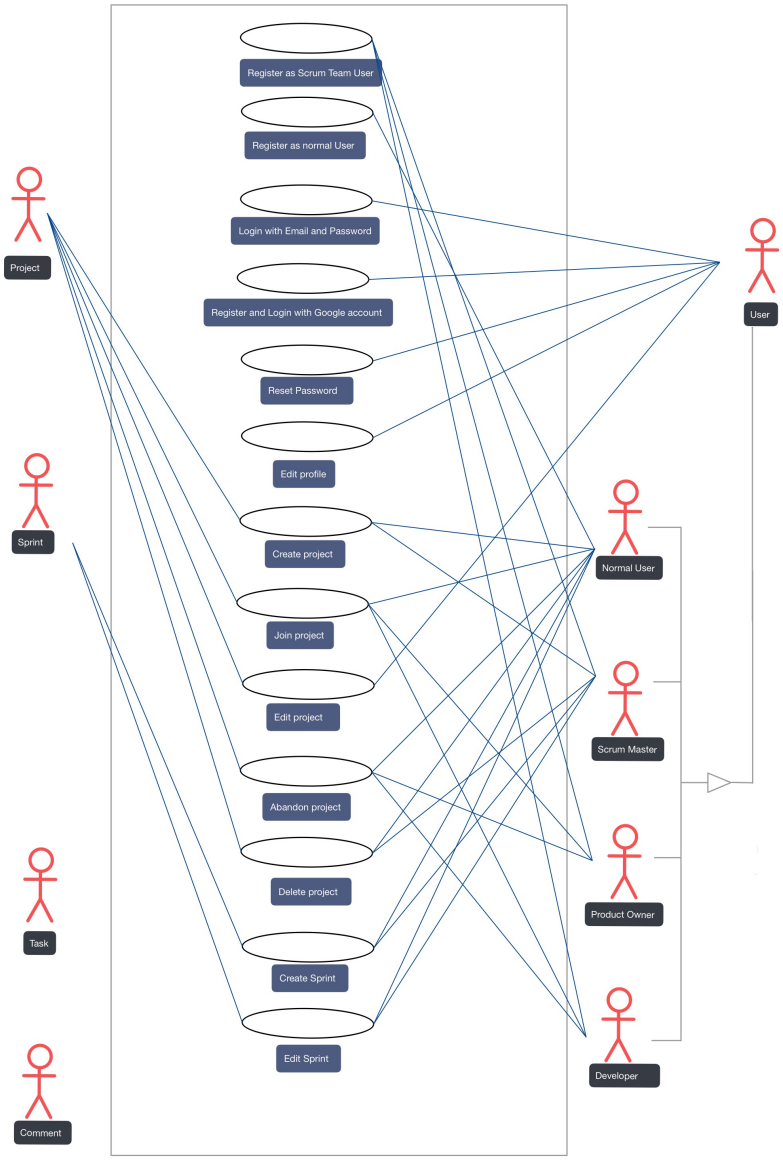


Figure 4.1: Use Case Model Part I

4.3 Analysis of each Use Case

Once we have established the use cases and the General Class Model, we analyze the attributes, methods and classes to use for each use case.

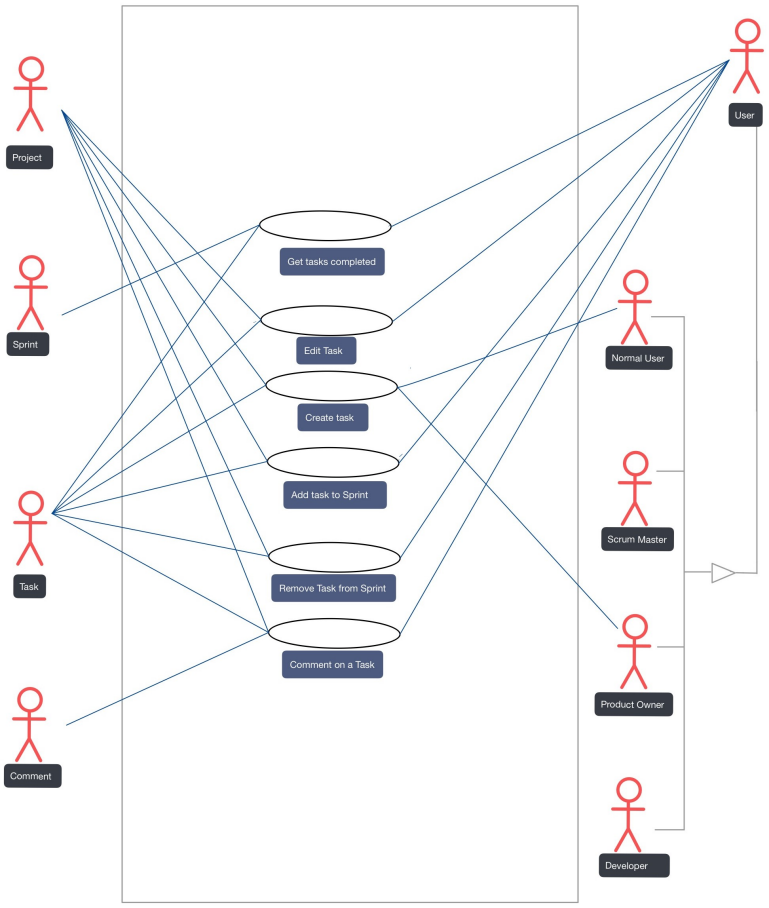


Figure 4.2: Use Case Model Part II

4.3.1 Register as Scrum Team User

The constructor of the classes *Scrum Master*, *Product Owner* or *Developer* is called with the user's *name*, *surname*, *email* and *imagePath* as parameters. For each of them, the *role* field is assigned.

4.3.2 Register as Normal User

The constructor of the class *NormalUser* is called with the user's *name*, *surname*, *email* and *imagePath* as parameters. The *role* field is assigned as "*normal*".

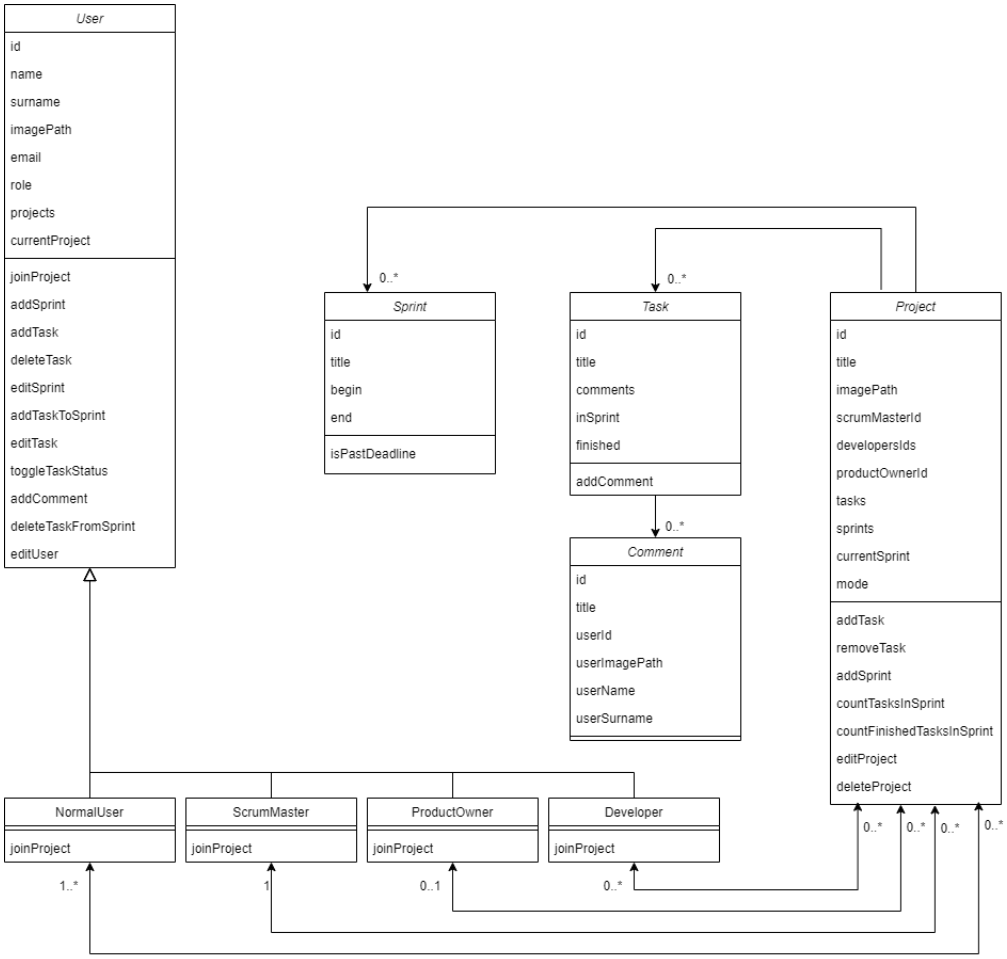


Figure 4.3: General Class Model

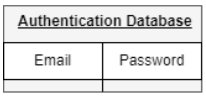


Figure 4.4: Database Model

4.3.3 Login with Email and Password

Once the user has been authenticate, the User's attributes from the class *User* are accessed.

4.3.4 Register and Login With Google account

The process on the database to write the email and password of the user changes, but for our model, it is the same process as previously described. We create the user's object when registering and we obtain the attributes values when logging in.

4.3.5 Reset Password

We update the database with emails and passwords.

4.3.6 Edit Profile

The *editUser* method of the class *User* is called, which has as parameters the new attributes values, which are updated.

Finally, the attributes *currentProject*, if needed, and *projects* of each member of the *projects*, where the user with the updated fields is a member, are updated.

4.3.7 Create Project

The constructor of the class *Project* is called with the *title*, *description* and *imagePath* as parameters. Also, the *id* of the Scrum Master or Normal user that created it is passed. We store it like this for the Normal User so we later know who the creator of the project is.

After that, the method *joinProject* from the classes Normal and Scrum Master User is called for the user that created it. In the *joinProject* method of the *normalUser*, the user's *id* is added to the *developersIds* arrray, as it is the array that will store the project members, are they all have the same role.

Then, for both Normal and Scrum Master User, the *currentProject* attribute is updated and the project is added to the array projects.

4.3.8 Join Project

The method *joinProject* is called for the classes Normal User, Product Owner and Developer. In the *joinProject* method of the *normalUser*, the user's *id* is added to the *developersIds* arrray, as it is the array that will store the project members, are they all have the same role.

In the *joinProject* method of the *productOwner*, the user's *id* is established as the *productOwnerId* attribute of the project.

In the *joinProject* method of the developer, the user's id is added to the *developersIds* arrray.

Then, for both Normal and Scrum Master User, the *currentProject* attribute is updated and the project is added to the array projects.

Finally, the attributes *currentProject* and projects of each member of the project are updated.

As we can see, for the work users, when creating the project, the Scrum Master *id* is established and when the Product Owner and Developers join, the product Owner *id* and the *developersId* arrays are updated. However, for the normal user, when creating the project, the creator user *id* is established (on the scrum Master *id* attribute of the project) and it is added as well to the *developersIds* array. When other users join, the *developersIds* arrays, which stores all of the members *ids*, is updated.

4.3.9 Edit Project

The *editProject* method of the class *Project* is called, which has as parameters the new attributes values, which are updated. The attributes *currentProject* and *projects* of each member of the project are updated.

4.3.10 Abandon Project

The *abandonProject* method of the class *User* is called, which has as parameter the project to abandon. First, the project's scrum Master *id*, product owner *id* and *developersIds* attributes are updated, depending on the user's *role*. Then, the attributes *currentProject*

and *projects* of each member of the project are updated. Finally, the *currentProject* attribute is updated to another project if there are more projects or to null if there are not any more projects for that user. Also, the attribute *projects* is updated.

4.3.11 Delete Project

The *deleteProject* from the class *Project* is called. This method can be called by the scrum Master of a project and by a normal user, only when the user is currently the only member of the project.

In the method, first, we remove the project from all of the members *projects* array and *currentProject* if needed. Then, we delete the project.

4.3.12 Create Sprint

The *addSprint* method of the class *User* is called, which has as parameter the sprint to add. First, the current project is updated with a new value for the attributes *sprints* and *currentSprint*. Then, the array projects is updated. Finally, the attributes *currentProject*, if needed, and projects of each member of the projects, where the user with the updated fields is a member, are updated.

4.3.13 Edit Sprint

This method can be called by the scrum Master of a project and by a normal user. The *editSprint* method of the class *User* is called, which has as parameters the new attributes values for the current sprint of the current Project, which are updated. The attributes *currentProject*, if needed, and projects of each member of the project are updated.

4.3.14 Get tasks completed

The *countTasksInSprint* and *countTasksFinished* methods of the class *Project* are called, which calculate the number of Tasks with the flag *inSprint* with the value true and the number of Tasks with the flag finished with the value *true*.

4.3.15 Edit Task

The *editTask* method of the class *User* is called, which has as parameters the task and the its new title. This method updates the title for that task, which is on the array *tasks* of the current Project. The attributes *currentProject*, if needed, and *projects* of each member of the project are updated.

4.3.16 Create Task

This method can be called by the product Owner of a project and by a normal user. The *addTask* method of the class *User* is called, which has as parameter the task to add. First, the current project is updated adding the task to the attribute *tasks*. Then, the array *projects* is updated. Finally, the attributes *currentProject*, if needed, and *projects* of each member of the projects, where the user with the updated fields is a member, are updated.

4.3.17 Add task to sprint

The *addTaskToSprint* method of the class *User* is called, which has as parameter the task to add to the sprint. First, the current project is updated, setting to true the value of the attribute *inSprint* of that task. Then, the array *projects* is updated. Finally, the attributes *currentProject*, if needed, and *projects* of each member of the projects, where the user with the updated fields is a member, are updated.

4.3.18 Remove task from sprint

The *removeTaskFromSprint* method of the class *User* is called, which has as parameter the task to remove from the sprint. First, the current project is updated, setting to false the value of the attribute *inSprint* of that task. Then, the array *projects* is updated. Finally, the attributes *currentProject*, if needed, and *projects* of each member of the projects, where the user with the updated fields is a member, are updated.

4.3.19 Delete Task

The *deleteTask* method of the class *User* is called, which has as parameter the task to delete. First, the current project is updated erasing the task from the attribute *tasks*. Then, the array *projects* is updated. Finally, the attributes *currentProject*, if needed, and *projects* of each member of the projects, where the user with the updated fields is a member, are updated.

5 Design

Once we have established the requirements and we have analyzed the use cases and class model for Scrumer, we need to design the architecture, the User interface and the logo of our App.

5.1 System Architecture

The system architecture design of the Scrumer app is based on a client-server model [31], utilizing various components to provide a seamless user experience. The architecture consists of four main entities: User, Authentication database, App Database, and Image Cloud Storage.

5.1.1 User

The User entity represents the end-users of the Scrumer app. Users interact with the app through a client-side interface, accessing various features and functionalities such as registration, login, and project management.

5.1.2 Authentication Database

The Authentication database entity is responsible for user authentication and authorization within the system. It stores user credentials (email and password) securely and handles the authentication process during login and registration. The Authentication database ensures that only authorized users can access the app and its functionalities.

5.1.3 App Database

The App Database entity serves as the central storage and synchronization mechanism for project-related data. It stores information related to users and projects: tasks, sprints and comments. The App Database enables real-time updates and synchronization across multiple clients, ensuring that all users have access to the latest project information and updates.

5.1.4 Image Cloud Storage

The Image Cloud Storage entity is responsible for storing and managing images (user profile and project) used within the app. It provides a storage infrastructure that allows users to upload, retrieve and display images related to their projects and their profiles. The Image Storing System ensures efficient and secure storage of images, facilitating their seamless integration within the Scrumer app.

To visualize the system architecture, a context diagram is created, illustrating the interactions and relationships between these entities. The context diagram provides an overview of the system's components and their connections.

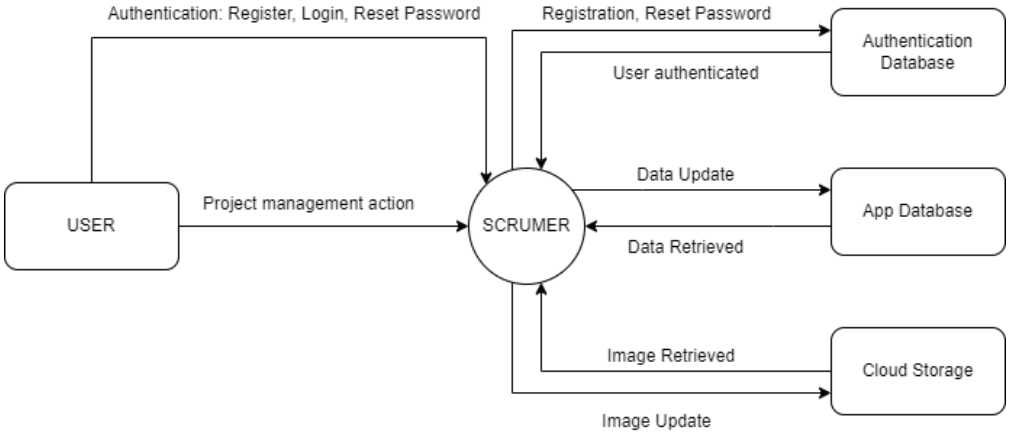


Figure 5.1: Design. Context Diagram

5.2 UI Design

The UI design for the Scrumer app includes various components and screens created using the App Miro tool for Ipad.[15] The color theme chosen is a light grey for the background and the color black is used for items in the UI. In addition, the color blue is used to represent actions related to the sprint.

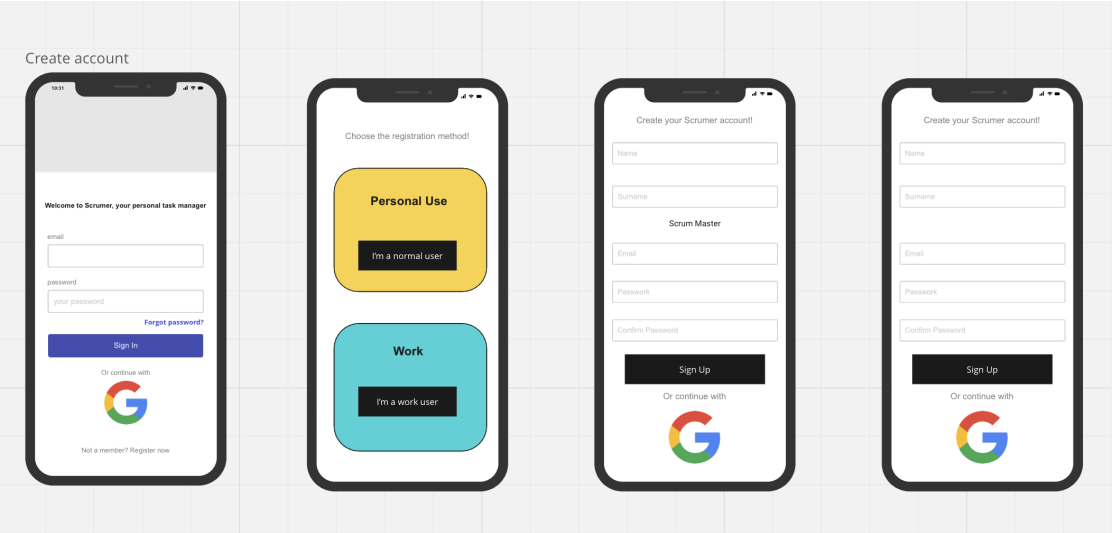


Figure 5.2: Design. Authentication UI

Regarding the use cases related to the Authentication Process, we design a UI that allows the user to login and to register. To login, the user enters the email and password, there is the possiblity to register and to reset password.

To register, the user can choose his/her type of user between normal User and the Scrum Team User. Then, depending on what he/she chose, the registration screen will be different.

For the scrum Team user, a scroll wheel allows him/her to choose his/her role, but for the normal user, it is only required the name and surname.

For both type of users, there are two possible methods to register and to login: using email and password or using a Google Account.

Overall, these designs aim to provide a seamless and intuitive user experience during the login and registration processes.

Once the user has logged in, for the navigation within the app, a navigation bar has been included. The navigation bar consists of four sections: Product Backlog, Sprint, Project, and Profile. Each section represents a different aspect of the app's functionality and contains several screens.

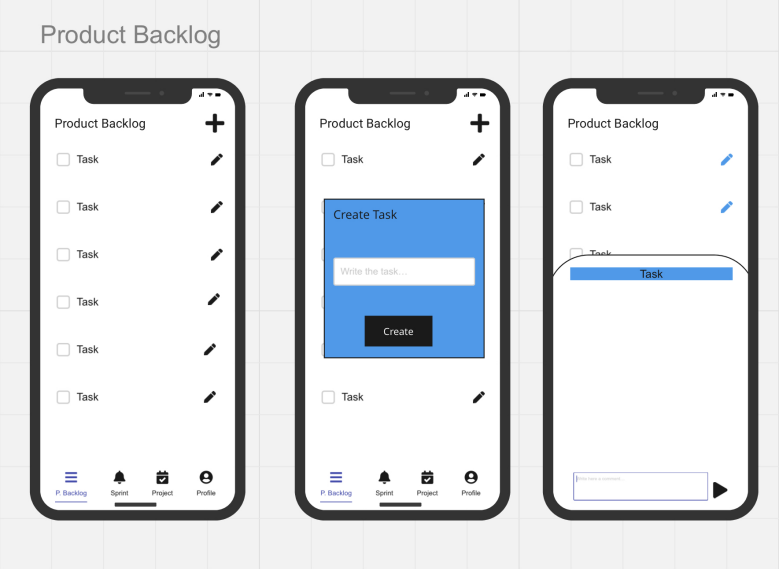


Figure 5.3: Design. Product Backlog UI

For the Product Backlog section, the UI design includes screens that allow users to view, edit and add tasks. Also, the design provides a bottom sheet for the comment section of each task.

The Sprint section incorporates screens that enable users to plan, track and review the project current sprint. It displays the tasks that are currently in the sprint and shows if the sprint is finished.

Also, it has a button, which provides a sprint report to the user, where the number of tasks finished is displayed and a new sprint can be created (it appears depending on the user type).

To create a sprint, a calendar and a clock is displayed to the user, to choose the date for the sprint. Furthermore, the sprint can be edited.

In the Project section of the app's UI, users can add projects: joining or creating a new one (depending on their user type) and view and edit the current project, having a section for the title, description, members, sprint report and image.

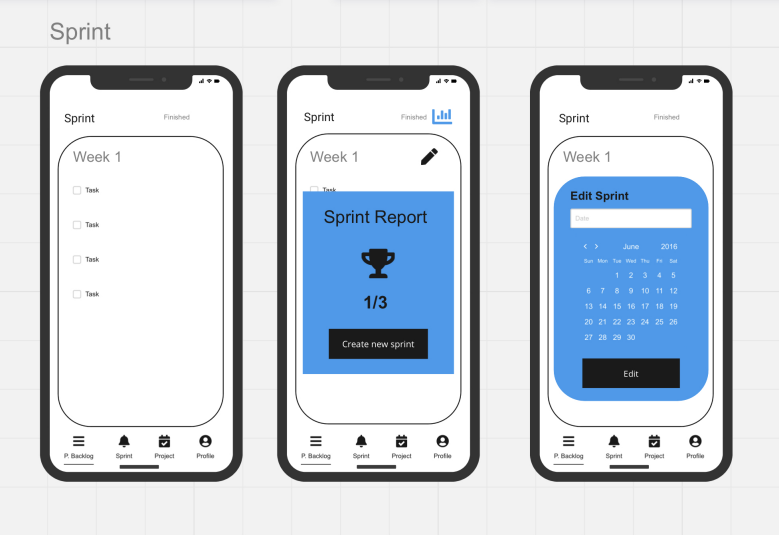


Figure 5.4: Design. Sprint UI

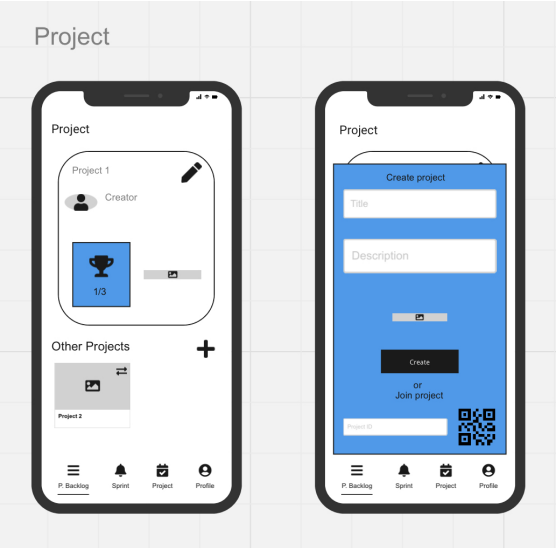


Figure 5.5: Design. Project UI

Lastly, the Profile section offers screens that allow users to manage their personal information. Users can view and update their profile details, and also,they can logout.

Overall, the UI design for the Scrumer app, created using the App Miro, focuses on delivering a visually appealing and intuitive user experience. The designs for the authentication use cases, along with the navigation bar and respective sections, provide a

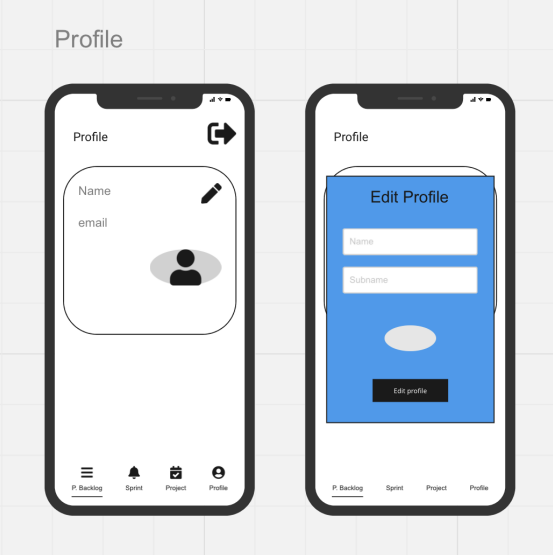


Figure 5.6: Design. Profile UI

cohesive and efficient interface for users to interact with the app's various features and functionalities.

5.3 Logo Design

The logo design for the Scrumer app presents a unique combination of elements that represent the concept of SCRUM project management, but it is also linked to the name of the App.

Taking inspiration from the association of a sprint with a rocket icon and the overall principles of SCRUM, the logo design aims to create a visually recognizable brand identity.

During the design process, various sketches and drawings are created to explore different possibilities and refine the logo concept. These initial iterations incorporate the rocket symbol and creatively integrated the letter SS"from Scrumer. The goal is to find a visually appealing composition that effectively communicates the app's focus.

After consideration and analysis, the final logo design is selected. It features a rocket shape seamlessly crafted from the letter SS"with the word SSCRUMERïnside the smoke of the rocket, making the Ü"from SSCRUMERä continuation to the drawing. This design

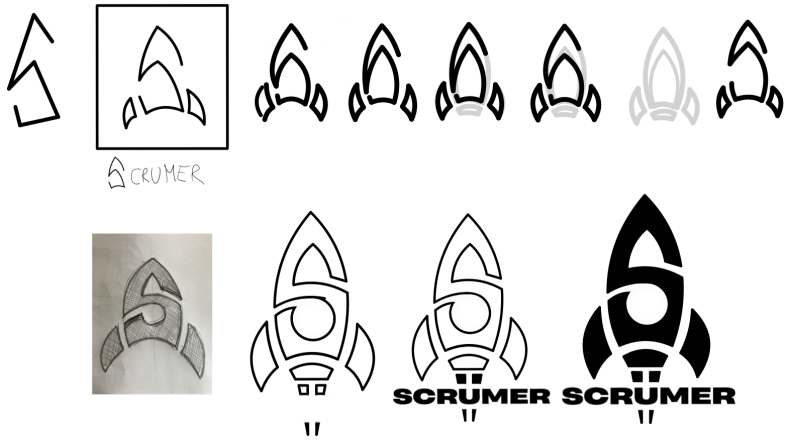


Figure 5.7: Design. Logo Sketches

not only captures the essence of Scrum but also establishes a strong association with the Scrumer app.

The logo predominantly uses the black color, which aligns with the theme that the app will have.

Figure 5.8: Design. Scrumer Logo

6 Implementation

This chapter covers all aspects related to the implementation of the App, including the build environment, the source code structure and the code implementation of each use case.

6.1 Build environment

The chosen development environment for the App is Visual Studio Code, which provides a robust and efficient platform for software development.

For the implementation of the App, two key technologies are utilized: Flutter, with its programming language Dart, and Firebase.

Regarding the system Architecture Design from the previous section, Flutter is used to implement the SCRUMER system and Firebase is used to implement the entities Authentication Database, App Database and Cloud Storage.[7]

6.1.1 Flutter

Flutter is a popular open-source UI framework developed by Google. It enables the creation of cross-platform applications with a single codebase, allowing for efficient development and consistent user experiences across different platforms.[8]

Therefore, Flutter allows us to deploy Scrumer in both IOS and Android devices.

The code written within the Flutter framework implements the system SCRUMER from the architecture design. Therefore, it implements the Class model established in the analysis section and use it to implement the App use cases, using the user interfaces designed on the Design section.

Furthermore, the Flutter App is linked to a Firebase project, which provides the systems for the Authentication Database, the App Database and the cloud Storage. At the end, we have the Flutter app Scrumer connecting the user with the firebase systems, following the previously established architecture design.

Dart

Dart is the programming language used in Flutter. It is a modern, object-oriented language with a strong type system, providing developers with the necessary tools to build applications.[4]

Widgets in Flutter

In Flutter, the code is structured around widgets, which are classes that describe the visual elements and behaviors of the application. [10] Widgets are configured using properties, which are parameters passed to the widget constructor. Properties allow us to customize the behavior, appearance, and content of the widget. Many widgets have properties to accept child widgets, allowing to nest and compose widgets to create complex user interfaces.

In Flutter, the structure of widgets follows a tree-like structure, where each widget can have child widgets. At the top level, Scrumer has the *MyApp* widget, which represents our entire application and is initialized in the main function. Within *MyApp*, we define the theme, title, and the home widget for our application: the *auth_page*.

Widgets in Flutter can either extend the *StatelessWidget* or *StatefulWidget* class. A *StatelessWidget* is a widget that does not have any mutable state. Once it is built, it remains the same throughout its lifetime.

On the other hand, a *StatefulWidget* is a widget that can change its internal state during its lifetime. It consists of two classes: the *StatefulWidget* itself and the corresponding *State* class, which contains the mutable state for the *StatefulWidget*.

In classes that extend either *StatelessWidget* or *StatefulWidget*, we define the *build* method.

The *build* method is a crucial method in Flutter widgets as it is responsible for creating and returning the widget hierarchy that represents the visual elements of your user interface. The *build* method is called automatically by the Flutter framework when it needs to build or rebuild the widget.

Inside the *build* method, we establish the widget tree by using different widgets provided by the Flutter framework, such as *Container*, *Text*, *Stack*, *Image*, *Button* and more. Additionally, we also utilize custom widgets defined in other classes to compose our Scrumer UI.

By nesting and configuring these widgets within the build method, we create the desired visual representation of our UI, following the design established previously.

6.1.2 Firebase

Firebase[5], a comprehensive mobile and web development platform, has been integrated into the App for various functionalities: Authentication database, App database and Image Cloud Storage. As the Flutter App is aimed for both IOS and Android Users, our firebase project does also support both operating systems.

Authentication Database

Firebase Authentication is utilized for user authentication and authorization purposes, ensuring secure access to the App's features and data.[6]

This database implements the entity Authentication Database [27] from the architecture design. Therefore, it stores the emails and passwords from the users and it is used for the registration and login of the users.

Firestore Database

Firestore Database is employed to store and synchronize real-time data across multiple devices and clients, enabling seamless data updates and collaboration.[2]

This database implements the entity App Database from the architecture design. Therefore, it stores the project-related data and user-related data and it is used for the project management actions.

On the Flutter App code, we work with the class model on Dart; but on firebase, the structure is different: the database works with Documents.

To resolve this issue, in Firebase, we have two different collection: Users and Projects. A User document is created when the user registers and a project document is created when a user creates a new project.

To correctly save and retrieve all of the user and project information, we have *toMap* and *fromMap* methods in all classes.

This way, for example, on the project creation, we first create the project object with its constructor and then, we call the *toMap* method. In this method, we create a map of its attributes; and for example, for its array attribute tasks, it calls the *toMap* method for each task in the array.

FireStore

Firebase Storage serves as a reliable solution for storing and retrieving images within the App, ensuring efficient image management and delivery.[3]

This database implements the entity Cloud Storage from the architecture design. Therefore, it stores the projects and user images and it is used for the project management actions.

On the class model for project and user, we have an attribute called *imagePath*, which is the path to the firebase FireStore location of the image.

6.2 Source Code Structure

The code structure of our Flutter app Scrumer follows a hierarchical architecture that promotes code organization, re-usability and maintainability.

The key components in the code structure are the *android*, *build*, *ios*, *lib* folders and the *pubspec.yaml* file.

- *android* folder: Android-specific code and resources that are necessary for building and running the app on Android devices.

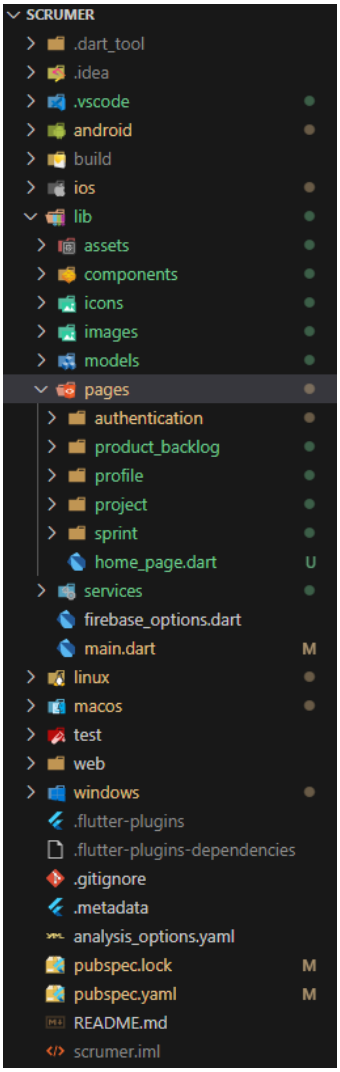


Figure 6.1: Code Structure

- *build* folder: It is automatically generated when we build or compile Scrumer. It contains the compiled and generated artifacts, as well as intermediate files, produced during the build process.
- *ios* folder: The platform-specific code and resources required for building and running the app on iOS devices.
- *lib* folder: The main codebase of the app. The key components of the *lib* folder are the *assets*, *icons*, *images*, *models*, *pages* and *services* subfolders and the *main.dart* file.

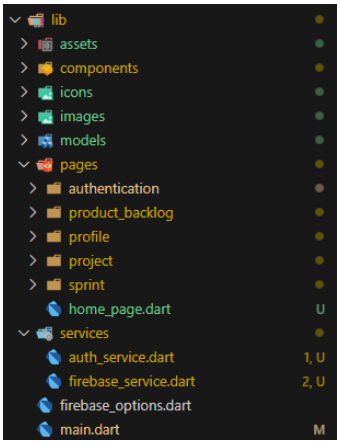


Figure 6.2: *Lib* folder Structure

- *assets* folder: This folder contains the app's lottie animations in the UI.
- *components* folder: This folder houses reusable UI components or widgets that can be used across multiple screens.
- *icons* folder: This folder holds the icon assets used in the app.
- *images* folder: This folder holds the images assets used in the app.
- *models* folder: This folder includes data models that represent the structured data used in Scrumer. The models correspond to the classes in the class model designed.

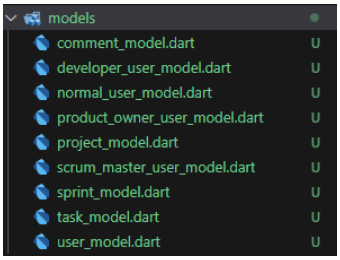


Figure 6.3: *Models* folder Structure

- *pages* folder: This folder contains individual screen widgets or UI components of the app. Each screen represents a distinct user interface view and is implemented as a Flutter widget. The folder is divided into subfolders corresponding to different sections of the app, such as Authentication, Product

Backlog, Sprint, Project, and Profile. Additionally, there is a separate screen for the *home_page*, which is used to build the navigation bar.

- *services* folder: This folder contains classes with methods responsible for handling the authentication process of the user and the connection for updating and retrieving data from the Firebase database. These services facilitate communication with Firebase and provide the necessary functionalities for data management.
- *main.dart*: This file contains the main() function, which sets up the app and specifies the Authentication page as the initial screen to be displayed. It serves as the entry point of the app and orchestrates the overall app structure.

- *test* folder: This folder includes a file with different unitary tests for the App.
- *pubspec.yaml*: This file is a configuration file that manages dependencies, assets, fonts, and other project-specific settings. It defines the required packages and resources used in Scrumer.

6.3 Save and Retrieve Data on Firebase

Several methods are implemented to achieve the process of saving and retrieving data faster and more efficiently. The idea behind this process is that we have our classes structure and we have the Firestore Database with two collections: users and projects, and the goal is to transform the data from one structure to the other easily.

The classes have the methods previously established on the class model, with the *toMap* and *fromMap* methods added. For the saving and retrieving data, a class with methods is implemented on the *firebase_service.dart* file on the *services* sub-folder of the *lib* folder. Also, some methods are added to the user and projects classes.

6.3.1 *toMap* and *fromMap* methods

To write data from the models structure to the Database, we add the methods *toMap*, which transform all of the attributes of a class into a map. To retrieve data from the Database to the models, we add *fromMap* methods, which transform the map obtained

from the Database into objects of the class model. For example, this is the fromMap project of the Product Owner User class:

```
static ProductOwnerUserScrumer fromMap(Map<String, dynamic> map) {
    return ProductOwnerUserScrumer(
      id: map['id'],
      name: map['name'],
      surname: map['surname'],
      email: map['email'],
    )
      ..role = map['role']
      ..currentProject = map['currentProject'] != null
          ? Project.fromMap(map['currentProject'])
          : null
      ..imagePath = map['imagePath']
      ..projects = (map['projects'] as List)
          .map((projectMap) => Project.fromMap(projectMap))
          .toList();
  }
```

Code snippet 6.1: *fromMap* project of the Product Owner User class

6.3.2 firebase_service.dart methods

The methods impleneted on the class *FirestoreService* on this file are: *get_user_class*, *setUserToCollection*, *setProjectToCollection*, *getUserById*, *getProjectById* and *deleteDocument*.

get_user_class method

This method is responsible for returning the corresponding object of the current user.

It first obtains the current user in the Authentication Database; secondly, it obtains the user document from the collection users on the Firebase Database, searching for the document with the User ID (obtained from the Authentication Database). Then, it accesses the user attributes that are structure as a map. Finally, depending on the role of the user, it calls the *fromMap* method to the corresponding user class: Normal, Scrum Master, Developer or Product Owner.

setUserToCollection method

This method is responsible for updating the user document on the Firebase Database.

It has as parameter the user object. It first access the document using the user ID and then, creates a map using the *toMap* method of the user. Finally, it saves the map on the user document.

setProjectToCollection method

This method is responsible for updating the project document on the Firebase Database.

It has as parameter the project object and it follows the same process as the *setUserToCollection* method.

getUserById method

This method is responsible for obtaining the user object that corresponds to an ID passed as an argument.

It first obtains a reference to the users collection and uses it to obtain the document, searching with the User ID. Then, it accesses the user attributes that are structure as a map. Finally, depending on the role of the user, it calls the *fromMap* method to the corresponding user class: Normal, Scrum Master, Developer or Product Owner.

getProjectById method

This method is responsible for obtaining the project object that corresponds to an ID passed as an argument.

It follows the same process as the *getUserById* method, creating a project object with the attributes of the database.

deleteDocument method

This method is responsible from erasing the document corresponding to the collection name and document ID passed as arguments.

It first obtains the collection instance and then, it calls the delete method for the document in that collection.

For all methods, when an exception occurs, an error message is displayed.

These method of the service folder are used on the classes user and project.

6.3.3 User Class

In the user class, for every method that needs to update the current user data, it calls *setUserToCollection(this)*. When the current project has been updated and we need to modify its database data, we have added new methods on the user class: *copyProject*, *updateProjectOfUser*, *updateObjectProjectInProjectsInMembers*,

updateProjectOfUser method

This method is responsible for updating the user document when a project of this user has been updated. The user and the project are passed as arguments.

It updates the *currentProject* field, if needed, and then, the project in the *projects* array.

updateObjectProjectInProjectsInMembers method

This method is responsible for updating the user document of all project members of the project passed as an argument.

It has another argument which corresponds with a flag *onAbandon* that is set to *false* and it will have the value *true* when the user has abandon the project.

It first updates the document of the project on the collection projects. Secondly, it updates the *currentProject* field, if needed, and then, the project in the *projects* array of this user, only when the user has not abandon the project.

Then, it updates the *currentProject* field, if needed, and then, the project in the *projects* array for all the project members, calling the method *updateProjectOfUser*.

copyProject **method**

This method is responsible for creating a clone of the project passed as an argument.

Now, when a method modifies data (adding a sprint, a task, removing a task, changing his/her name, etc) from the current project: it first copies the project, modifies the copy and then calls the method *updateObjectProjectInProjectsInMembers* to update it in the database. In the *abandon* method, after doing that process, it removes the project from the *projects* user array and changes the value of the *currentProject* to *null* if there are no other projects or to the first project. Finally, it updates the user document calling *setUserToCollection*.

6.3.4 Project Class

In the project class, for the methods *editProject* and *deleteProject*, we have added the method *updateValues* and *updateValuesBeforeDeleting*,

updateValues **method**

This method is responsible for updating the project attributes *title*, *description* and *imagePath* for a user passed as an argument.

It first updates the *project* array and then, if needed, the *current project*attribute. Finally, it calls the method *setUserToCollection* to update that user document.

Therefore, in the *editProject* method, it first accesses each user member with the method *getUserById* passing the *scrumMasterId*, *productOwnerId* or each ID from the *developersIds* attributes of the project as arguments. Then, it calls the method *updateValues* for each member. Finally. it calls *setProjectToCollection*.

updateValuesBeforeDeleting method

This method is responsible for removing the project from the projects array of user passed as argument and then, setting the current project to null if there are no other projects or to the first project. After, it updates the user document calling *setUserToCollection*.

Therefore, on the *deleteProject* method, it calls the *updateValuesBeforeDeleting* method for each project member and then, it calls the *deleteDocument* method.

6.4 Refresh Data from Database

When a change happens on the database, we need to update the interface for all users affected by the change. To achieve this, in the pages needed, we have two different methods: an initializing retrieve method and a subscription to updates.

6.4.1 Initial Retrieve method

To display the data for the page for the first time, we first have a flag called *flagUpdateNeededInitial* with the value *true* and a *userScrumer* variable; then, we create a method called refreshData, which is an *async* method. This method assigns the value of the object user to the *userScrumer* variable, calling the *get_relevant_fields* method. Finally, it sets the value of *flagUpdateNeededInitial* to *false*.

On the *build* method, while the *refreshData* method has not update the value of *flagUpdateNeededInitial* to *false*, it displays a *CircularProgressIndicator* and when the *userScrumer* value has been assigned, the page is built.

6.4.2 Subscription to updates

To update the UI when a change is made, we first have a flag called *subscriptionStarted* set to *true* and *StreamSubscription* variable named *_userSubscription*. [14]

Then, we create a method called *_subscribeToUserUpdates,*. This method sets *_userSubscription* to listen to changes on the database and when a field appearing on the UI is changed, a method that handles data changed is called. This method that handles data

changed calls the *refreshData* method. Finally, it sets the value of *subscriptionStarted* to *false*.

Finally, we also create the method *_unsubscribeFromUserUpdates*, which cancels the subscription and it is called on the *dispose* method.

On the *build* method, if the *subscriptionStarted* flag is false, it calls the method *_subscribeToUserUpdates*.

6.5 Responsive Implementation

To have a responsive implementation, and build an app that adjusts to all screen sizes, the *MediaQuery.of(context).size* property is used. [1]

In the pages needed, the width and the height of the screen is obtained using *MediaQuery.of(context).size.width and MediaQuery.of(context).size.height* and they are used for the establishing the sizes of the different widgets in the page. For example, the Scrumer logo on the login page needs to be proportionate to the screen size and to adjust when a phone is on landscape mode:

```
double logoSize = screenWidth > screenHeight
    ? screenWidth * 0.3
    : screenHeight * 0.3;
```

Code snippet 6.2: Calculation of logo size based on screen dimensions

6.6 Authentication

The first step in the implementation is the authentication part, which includes the Registration and the Login. On the folder *lib*, we have a sub-folder which includes all files with the classes corresponding to the widgets for the authentication. The widgets that correspond to actual pages the user interacts with are: the *login page*, the *choose registration page*, the *register normal user page*, the *register work user page* and the *reset page*.

The *auth page* and *login or choose registration page* are used to navigate within this pages. And the *role wheel scroll widget* is used within the *register work user page*.

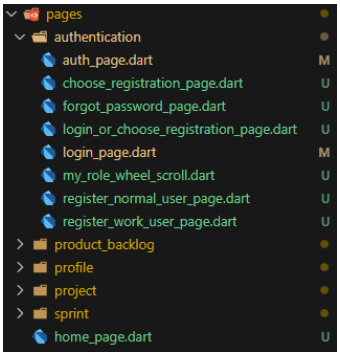


Figure 6.4: *Authentication* sub-folder Structure

Auth page

When the user opens the App, the initial screen is the *auth_page*. The *authPage* widget extends from a *StatelessWidget* and it checks if the user is already logged in or not.

To check if the user is already logged in, we utilize the firebase Authentication Database. We use the *StreamBuilder* widget, which listens to changes in the authentication state.[26]

The *StreamBuilder* takes two parameters: the stream and the builder. The stream parameter is set to *FirebaseAuth.instance.authStateChanges()*, which provides the stream of user authentication state changes. This stream provides updates whenever the user's authentication status changes. The builder parameter is a callback function that gets called whenever a new value is received from the stream. It takes two arguments: the context and the snapshot of the stream data. The snapshot object represents the latest state of the stream. By accessing *snapshot.hasData*, we can check if the user is currently logged in or not.

If the user is logged in, it goes directly to the *HomePage*, without making the user login again. Once the user has login or register, it will come back to this page and, as the snapshot will have data (it will have just been updated), it will also go to the *HomePage*. If the user isn't logged in, it returns the widget *LoginOrChooseRegistrationPage*.

LoginOrChooseRegistration page

The *LoginOrChooseRegistrationPage* widget is responsible for switching between the screen login and registration. In Flutter, the widgets work as a stack, which means that one goes on top of the other; to avoid the login and the registration screens to do this, when the user navigates from one to the other, we utilize this widget.

The *LoginOrChooseRegistrationPage* widget extends from the *StatefulWidget* and therefore, has a state. This state corresponds with a flag called *showLoginPage* which gets updated on the method *togglePages*. This method gets called every time the user press the option to go to the *login page* on the *choose registration page* and viceversa.

At the end, depending on the value of this flag, the build method returns the *LoginPage* or the *ChooseRegistrationPage*.

6.6.1 Authentication Service: Authenticate with Google

The *AuthService* class on the *services* subfolder of the *lib* folder handles the user authentication with his/her Google Account. It uses the Authentication Database.

We use this class for both the Registration and the Login, therefore, the class has a constructor that takes a boolean parameter *isRegistrationPage* to indicate whether the current page is a registration page or not.

The class has a *_auth* field, which is an instance of *FirebaseAuth* from the *firebase_auth* package, which is used to interact with our Firebase Authentication Database.

The class has a method called *checkIfEmailExists*, which is used to check if an email already exists in the Firebase Authentication database. It takes an email as a parameter, and internally it calls *fetchSignInMethodsForEmail* to retrieve the sign-in methods associated with the given email. If the sign-in methods list is not empty, it indicates that the email exists in the authentication database.

The class has a method called *signInWithGoogle*, which handles the sign-in process with Google.

It begins the interactive sign-in process by calling *GoogleSignIn().signIn()*, which opens a Google sign-in dialog.

Once the user has introduced his/her google account data, it calls the *checkIfEmailExists* method.

Depending on the result from the *checkIfEmailExists* result and the value of *isRegistrationPage*, the method returns different values: *0* if the email does not exist and it's not a registration page, indicating the need for registration, or *1* if the email exists and it is a registration page, indicating that the user should proceed with the registration process.

If the email exists and it's not a registration page, or if the email does not exist and it's a registration page, the method continues by obtaining the authentication details from the Google sign-in request.

It creates a new credential using *GoogleAuthProvider.credential* with the obtained authentication details.

Finally, it calls *FirebaseAuth.instance.signInWithCredential(credential)* to sign in with the credential and returns the result of the sign-in process.

6.6.2 Login Page

The Login Page extends from a *StatefulWidget* as it has as a state the email and password text editing controllers, which have as values the text for the email and password of the user to login in.

SignUserIn **method**

It has a method called *signUserIn* which handles the process of signing a user into the application. It also includes a *showErrorMessage* method to display an error message dialog if an error occurs during the sign-in process.

This method attempts the sign-in process. It uses *await* and the *signInWithEmailAndPassword* method provided by *FirebaseAuth.instance* to authenticate the user. The email and password values are obtained from the *emailController* and *passwordController* respectively, and trimmed using *trim()*, to erase the white spaces the user might have written. For the whole process, a *Circular Progress Indicator* inside of a *Dialog* is being showed to the user.

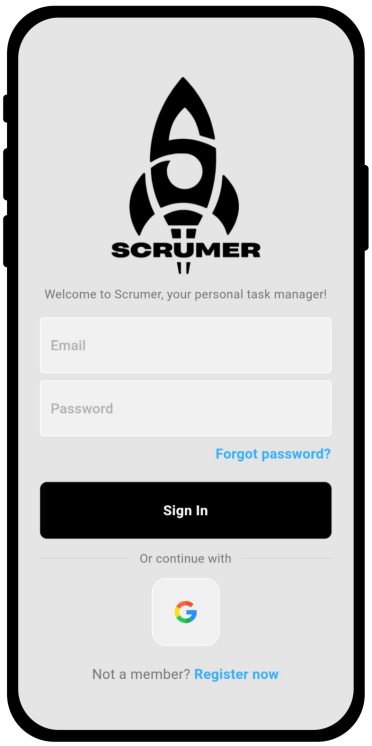


Figure 6.5: Login UI

If the sign-in process is successful, the loading dialog is closed. This is achieved by calling *Navigator.pop(context)*, which removes the top route (in this case, the loading dialog) from the navigation stack.

If an exception of type *FirebaseAuthException* occurs during the sign-in process, the loading dialog is closed as before, and the *showErrorMessage* method is called, passing the error code *(e.code)* as the parameter.

The *showErrorMessage* method displays an *AlertDialog* to the user, showing the provided error message (message) in the dialog's title, which can be: 'Email needed', 'Password needed', 'User not found', etc.

User Interface

The 'build' method creates the login page in the Scrumer application according to the design previously created.

The *Scaffold* widget is used as the base structure for the page. The *backgroundColor* property is set to *Colors.grey[300]* to give it a light grey background. Inside, a *SafeArea* widget, used to ensure that the content is visible and not obscured by system elements like notches or status bars, that contains a *SingleChildScrollView* widget that is used to enable scrolling when the content exceeds the screen size.

Finally, the elements of the UI are inside a *Column*, used to vertically stack multiple child widgets. The main elements of the User Interface are:

- An *Image.asset* widget used to display the Scrumer logo.
- A *Text* widget used to display a welcome message.
- Two *MyTextField* widgets are used to display email and password text fields with its corresponding controllers properties set. This custom text field widget is implemented on the components folder.
- A *Row* widget used to display the 'Forgot password?' text, when the text is pressed, the *Navigator.push* method is called to navigate to the *ForgotPasswordPage*.
- A *MyButton* widget (from the components folder), used to display the 'Sign In' button. The button has two properties: *text* and *onTap*. The *text* property is set to 'Sign In', and the *onTap* property is set to the *signUserIn* method, which handles the sign-in process.
- Another *Row* widget used to display a divider line and the 'Or continue with' text.
- A *SquareTile* widget to display the Google sign-in buttons. The *SquareTile* is a custom widget established on the components folder and it provides an *onTap* callback and an image path (in this case we pass the google logo image) using a *GestureDetector* for a square *Container*, with the image as its child. Inside the *onTap* function, asynchronous code is executed to perform the sign-in process with Google.

  It utilizes the *AuthService* class, passing *isRegistrationPage: false*, and invokes the *signInWithGoogle()* method to initiate the Google sign-in process.

  If the *signInWithGoogle()* method states that the email of the user does not exist, a dialog with the message 'Registration needed is displayed'.

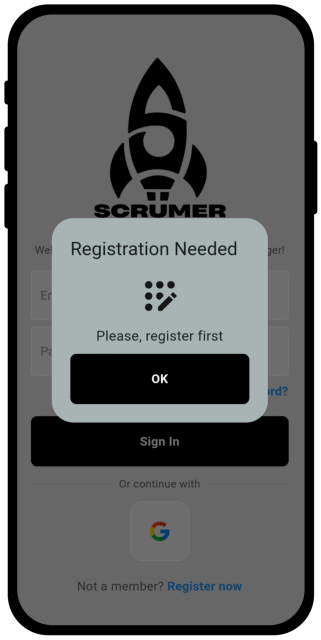


Figure 6.6: Login: Register First message

- Finally, another *Row* widget is used to display the 'Not a member?' and 'Register now' text. The 'Register now' text is wrapped with a *GestureDetector* widget, which triggers the *onTap* callback provided via the widget onTap property. On the *onTap*, it executes the method *togglePages* from the *LoginOrChooseRegistrationPage* widget and navigate to the *choose registration page*.

6.6.3 Choose Registration Page

The Choose Registration Page extends from a *StatefulWidget* and it is responsible for displaying the interface for the user to choose between the two registration methods: normal user or work user.

User Interface

The 'build' method creates the choose registration page in the Scrumer application according to the design previously created.

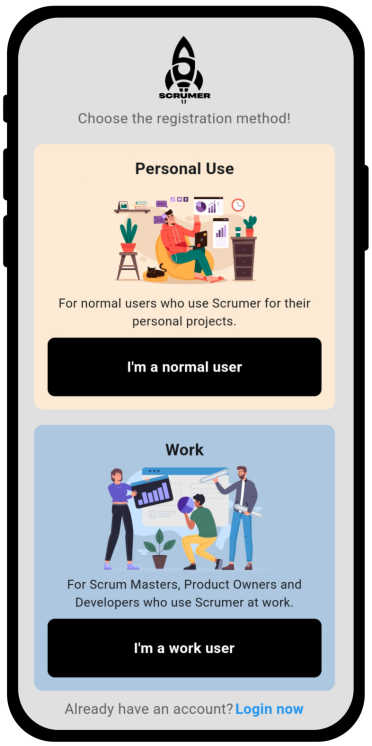


Figure 6.7: Choose Registration UI

It has a similar widget tree as the login, having a *Scaffold*, a *SafeArea*, a *SingleChildScrollView* and a *Column* nested.

The main elements of the User Interface, which are children of the column, are:

- An ‘Image.asset‘ widget used to display the Scrumer logo.
- A ‘Text‘ widget used to display a message instructing the user to choose the registration method.
- Two registration option containers:
  - First Container: Represents the "Personal Use"registration option. It contains a title text, a Lottie animation (loaded from an asset on the *assets* folder), a description text, and a MyButton widget to handle the registration process for normal users.

  - Second Container: Represents the "Work"registration option. It contains a title text, a Lottie animation, a description text, and a MyButton widget to handle the registration process for work users.

- Finally, another ‘Row‘ widget is used to display the Älready have an account?änd "Login now"text, with a *GestureDetector* that executes the method *togglePages* from the *LoginOrChooseRegistrationPage* widget and navigates to the *login page*.

6.6.4 Reset Password Page

The Reset Password Page extends from a *StatefulWidget* as it has as a state the email text editing controller, which have as value the text for the email of the user.

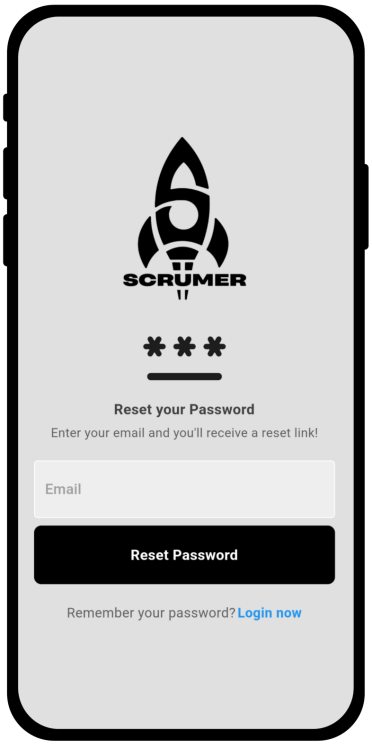


Figure 6.8: Reset Password UI

User Interface

The *build* method creates the choose registration page in the Scrumer application according to the design previously created.

It has a similar widget tree as the login, having a *scaffold*, a *safeArea*, a *SingleChildScrollView* and a *Column* nested.

The main elements of the User Interface, which are children of the column, are:

- An *Image.asset* widget used to display the Scrumer logo.
- A *Text* widget used to instructing the user to enter their email to receive a reset link.
- A *MyTextField* widget that allows the user to enter his/her email.
- A Row widget used to display the "Forgot password?"text, when the text is pressed, the *Navigator.push* method is called to navigate to the *ForgotPasswordPage*.
- A *MyButton* widget (from the components folder), used to display the "Reset Password"button. The button has two properties: text and onTap. The ‘text‘ property is set to "Reset Password", and the *onTap* property is set to the *passwordReset* method, which handles the reset password process.

  PasswordReset uses the *FirebaseAuth.instance.sendPasswordResetEmail*method to send a password reset email to the user's entered email address. If the email is sent successfully, it displays a success message using *showMessage*. If an error occurs, it displays an error message.
- Finally, another *Row* widget is used to display the "Remember your password?änd "Login now"text, allowing users to navigate back to the login page using the *Navigator.pop* method.

6.6.5 Registration for work user

The Register work user Page extends from a *StatefulWidget* as it has as a state the *email*, *password*, *confirmPassword*, *name* and *surname* text editing controllers. It does also have a *FirebaseFirestore* variable and a *selectedRole* variable.

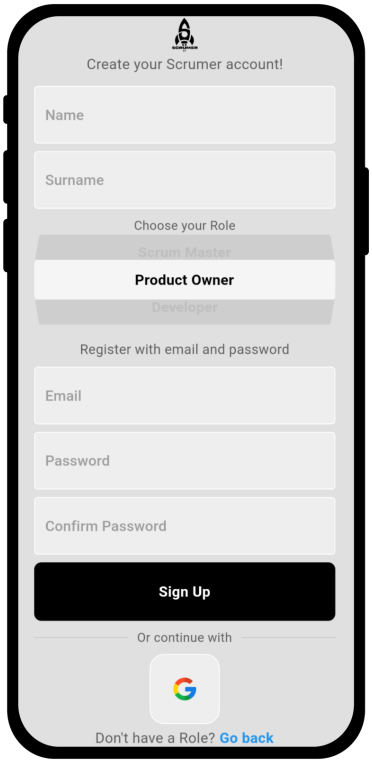


Figure 6.9: Register Work User UI

signUserUp method

It has a method called *signUserUp* which handles the user registration process in an application. It performs several checks and validations before creating a new user in our Firebase Authentication and storing user data in our Firebase Database users collection.

The method starts by checking if the entered password matches the confirmed password. If they don't match, it calls the *showErrorMessage()* method to display an error message stating "Passwords don't match!". Then, it checks if the name, surname, and selected role (such as SScrum Master,Product Owner,ör "Developer") are filled in; if not, it calls the *showErrorMessage()*.

Then, it attempts to create a new user in Firebase Authentication using the *createUserWithEmailAndPassword()* method.

Once the entry of the user has been created on the Authentication Database, depending on the selected role, a corresponding *UserScrumer* subclass instance is created. The user's unique identifier (user.uid), name, surname, and email are passed to the constructor of the appropriate subclass.

Finally, the user object is converted to a map using the *toMap()* method and stored in Firestore under the üsers"collection with the user's unique identifier as the document ID.

This method attempts the sign-up process. It uses *async* and the *signInWithEmailAndPassword* method provided by *FirebaseAuth.instance* to authenticate the user. The email and password values are obtained from the *emailController* and *passwordController* respectively, and trimmed using *trim()*, to erase the white spaces the user might have written. For the whole process, a *Circular Progress Indicator* inside of a *Dialog* is being showed to the user.

If the sign-up process is successful, the loading dialog is closed. This is achieved by calling *Navigator.pop(context)*, which removes the topmost route (in this case, the loading dialog) from the navigation stack.

If during the process, an exception occurs, the *showErrorMessage* method is called and it displays an *AlertDialog* to the user, showing the provided error message (message) in the dialog's title, which can be: 'Email needed', 'Password needed', 'User not found', etc.

User Interface

The *build* method creates the choose registration page in the Scrumer application according to the design previously created.

It has a similar widget tree as the login, having a *Scaffold*, a *SafeArea*, a *SingleChildScrollView* and a *Column* nested.

The main elements of the User Interface, which are children of the column, are:

- An *Image.asset* widget used to display the Scrumer logo.
- A *Text* widget used to instructing the user to create his/her Scrumer Account.
- Two *MyTextField* widget that allows the user to enter his/her name and surname.

- A *Text* widget used to instructing the user to choose his/her Role, followed by a *RoleWheel* widget [], which is a custom widget fro the components folder which allows the user to scroll through the three roles and choose one. When the user has selected the role, the selected role is updated.
- A *Text* widget used to instructing the user to type the email and password followed by three *MyTextField* widget that allows the user to enter his/her email and password twice.
- A *MyButton* widget (from the components folder), used to display the SSign Up"button. The button has two properties: *text* and *onTap*. The *text* property is set to "Reset Password", and the *onTap* property is set to the *singUserUp* method, which handles the signing up process.
- Another *Row* widget used to display a divider line and the Ör continue with"text.
- A *SquareTile* widget to display the Google sign-in buttons. The *SquareTile* is a custom widget established on the components folder and it provides an *onTap* callback and an image path (in this case we pass the google logo image) using a *GestureDetector* for a square *Container*, with the image as its child. Inside the *onTap* function, the *signUpwithGoogle* method is called.

  In the signUpwithGoogle method, several checks regarding the name, surname and role take place, as in the normal *singUp* method. Then, it utilizes the *AuthService* class, passing *isRegistrationPage: true*, and invokes the *signInWithGoogle()* method to initiate the Google sign-up process. Once, the Google sign-up process continues with the same process as the normal *singUp* method: it creates an object for the corresponding user class and it maps to write the user data in a new document on the *users* collection of the database.
- Finally, another *Row* widget is used to display the "Don't have a Role?änd "Go back"text. The "Register now", allowing users to navigate back to the choose registration page using the *Navigator.pop* method.

6.6.6 Registration for normal user

The registration for the normal user has a similar implementation as for the work user.

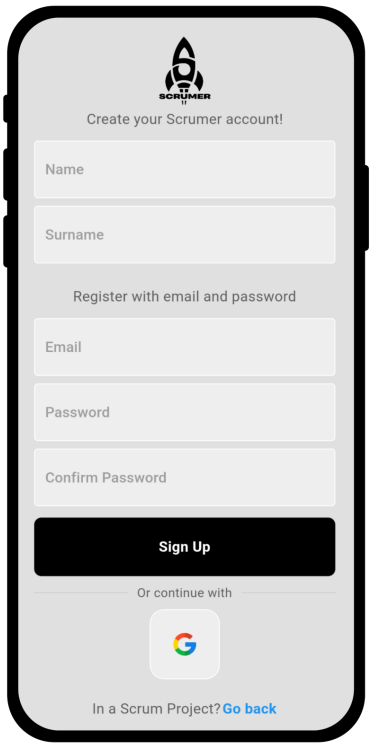


Figure 6.10: Register Normal User UI

However, the normal users do not have scrum roles, so the feature to choose a role does not appear; therefore, all of the code referring to the selection of the role does not exists and the sizes and spaces between widgets is different as well.

With this implementation, all of the use cases and goals regarding the authentication process have been implemented and achieved.Also, the system architecture has been established, having written the user email and password on the Authentication Firebase Database and the User data (mapped from the class model) on the collection users of Firebase Database.

6.7 Home Page

The Home Page represents the main content and navigation logic for the home page of the application when the user is logged in.

The *WidgetsBinding.instance.addPostFrameCallback* is used to schedule a callback after the current frame has finished rendering. In this case, it is used to remove the focus from any input fields by calling *FocusScope.of(context).unfocus*. This ensures that when the page is loaded, no input fields, from for example, the login page, have focus.

In this page, the bottom navigation bar of the App is implemented using the custom MyBottomNavBar widget [29] from the *components* folder. We have four main pages corresponding with each section of the navigation bar: Product Backlog, Sprint, Project and Profile. The currently selected page is stored in the *_selectedIndex* variable, which is updated when the user taps on a navigation item through the *navigateBottomBar* method. The *_pages* list holds the different pages that correspond to each navigation item.

In the *build* method, the *Scaffold* widget is used as the main layout container. It has three properties: body, backgroundColor and bottomNavigationBar.

The *body* property is set to the value of the *_selectedIndex* of the *_pages* list, which displays the currently selected page. The *backgroundColor* property of the Scaffold is set to Colors.grey[300] to provide a light grey background. The *bottomNavigationBar* property is set to the *MyBottomNavBar* widget, which handles the navigation between different pages.

6.8 Project

For the implementation of the project section of the Navigation Bar, the sub-folder project includes several widgets. The widget that at the end, is responsible to display the whole project page is the ProjectPage widget.

6.8.1 *ProjectPage*

This is the widget responsible for displaying all of the UI related to the Project.

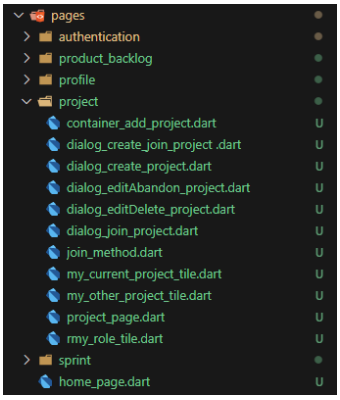


Figure 6.11: *Project* sub-folder Structure

Project UI

The UI consists of a *ListView*, wrapped with a RefreshIndicator[21], with provides a pull-to-refresh functionality to update (calling the method *refreshData*) its content when the user performs a swipe-down gesture. The *ListView* has various components:

- A *Row* with the logo of Scrumer, with the modification of having the "PROJECT"text instead of SCRUMER, and the Project title.
- The *CurrentProjectTile* widget, which displays the information and status of the current project.
- If the user has other projects, it displays the Öther Projects"title and an Ädd"button if there is more than one project.
- If the user has other projects, an horizontal *ListView.builder* to display the *OtherProjectTile* widgets for each project, allowing the user to switch between projects.

6.8.2 *CurrentProjectTile* widget

The *CurrentProjectTile* widget displays the details of the current project for a user. It includes the project's title, description, team members, the sprint information and an image. It also provides options to edit, delete, and abandon the project.

There are two possible cases:

1. The user does not have projects. In this case, the *ContainerAddProject* is return.

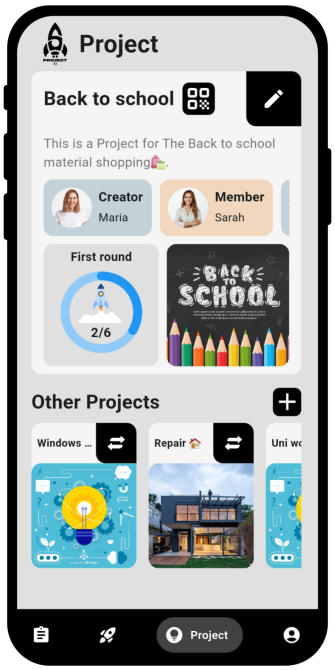


Figure 6.12: Project UI

2. The user has at least one project. In this case, a *Container* with the project's title, QR button, edit button, description, team members, sprint report and image is returned.

QR Button

The QR button opens a *showModalBottomSheet*[9], where the QR linked to the Project ID is shown using a *QrImageView*, from the package *qr_flutter*[24], which has been added to the dependencies on the *pubspec.yaml* file.

Team Members UI

For the team members data, several widgets and variables are used. Firstly, there are several variables for the names of the Scrum Master, product Owner and developers. Also, they have their respective *imagePath* variables.

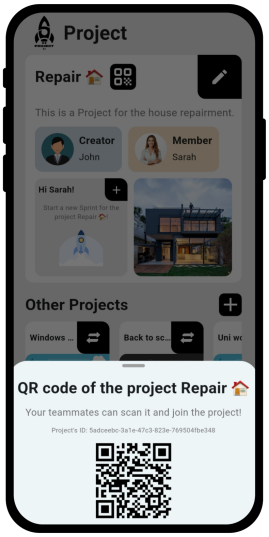


Figure 6.13: Project QR UI

The *CurrentProjectTile* has the *getName* and *getImagePath* methods, which obtain the user object corresponding to the ID string passed as an argument, and then, they return the user nae and the user *imagePath*. If they do not find any user with that ID, the method returns 'Nobody' as the name. And when the user has a default profile image and it is a work user, the method returns the default *imagePath* for the user role: there are different default images for the Scrum Master, Product Owner and Developers.

The *CurrentProjectTile* has the *refreshDataNames* method, which receives as parameter a String scrumMasterId, a String productOwnerId and a list of Strings developersIds. This method updates the values of the names and imagePaths variables calling the *getName* and *getImagePath* methods for each ID.

When the widget is initialize or an update happens, the *refreshDataNames* method is called.

To display the members data, a *Container* widget is used . It contains a *ListView.builder* that creates a list of items dynamically based on the number of team members.

The *itemCount* of the L*istView.builder*is determined by the user's role and the number of developers in the project. If the user is not a normal user, the count is set to 2 (ScrumMaster and Product Owner) plus the number of developers. Otherwise, it's set

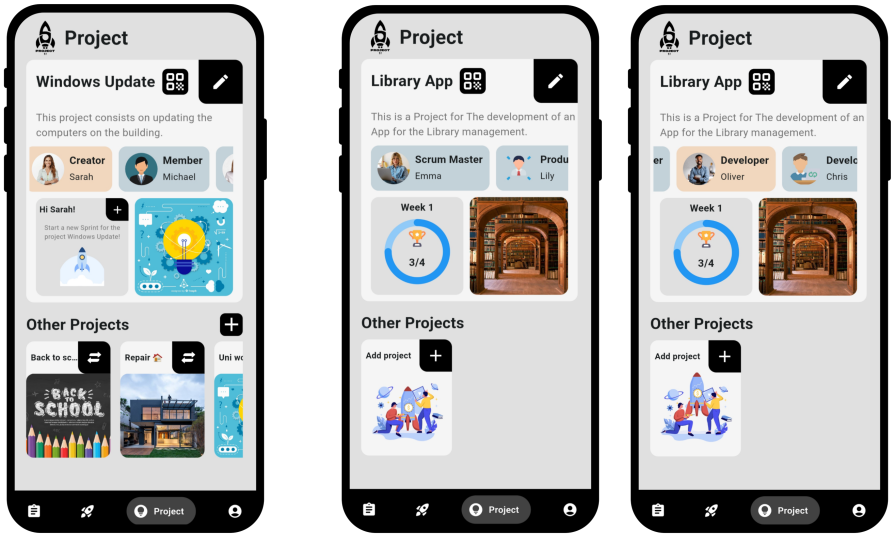
to the number of developers only, as on the normal mode, we have all members of the project saved on the developersIds array.

Inside the *itemBuilder*, the individual team members are created based on their index. The first two items are for the Scrum Master and Product Owner and are only displayed if the user role is not "normal", while the rest are for the Developers.

To display in a special way the current user, we have a flag *itisUser* which is set a true after checking for the actual user.

To display each member information, the *RoleTile* widget is returned. On The *RoleTile* widget takes four required parameters: *name*, *rolepath*, *role*, and *itIsUser* (we pass the flag we have update it stating if that user is the current user).

For the the "work"mode, it first displays the Scrum Master and Product Owner *RoleTile* passing the variables names and *imagePaths* established with the *refreshDataNames* method, with the Scrum roles. When there is no product Owner on the team, an empty *Container* is returned. Then, it displays the Developers *RoleTiles*.



(a) Normal User (b) Work User

Figure 6.14: Project Team Members UI

For the "personal"mode, the members are in the *developersIds* list, so it displays the Developers *RoleTiles*. For the role, instead of developer as a role, the word 'Member' is passed, except for the creator of the project, for whom the *Role* is 'Creator'. We know

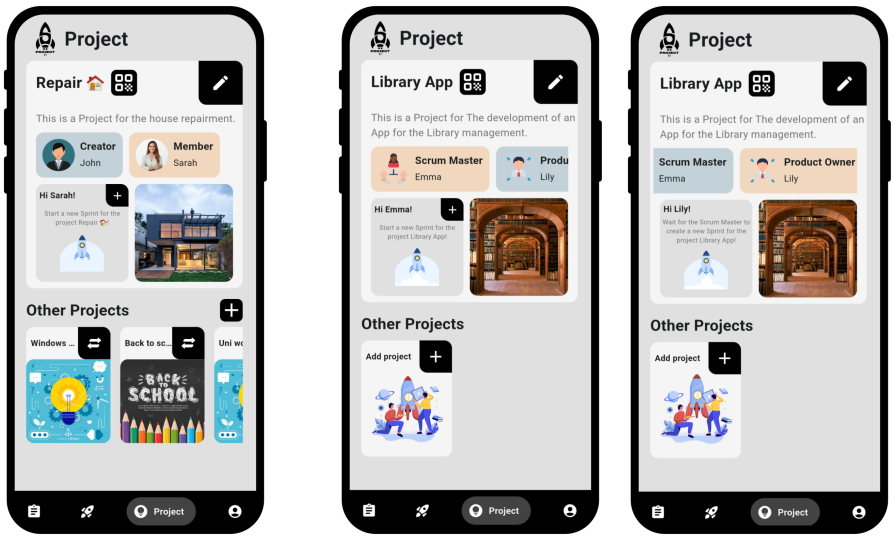
who the creator is because as we stated on the class model design, his/her ID is saved on the *ScrumMasterId* attribute of the project.

RoleTile widget

The *RoleTile* widget is a custom widget that displays the information of a team member in a stylized container. The widget takes four required parameters: *name*, *rolepath*, *role*, and *itIsUser*.

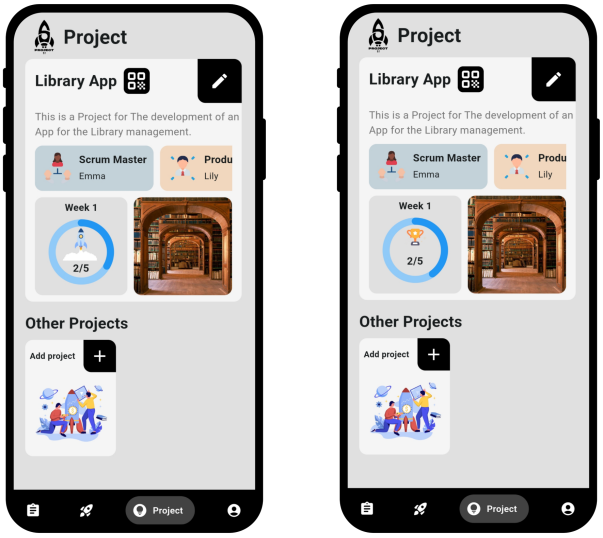
Inside the build method, a Container is used, with background color depending on whether the team member is the current user (*itIsUser*). The content of the container is composed of a *Row* widget that contains an image and text. The image is displayed using the *ClipRRect* widget to apply rounded corners. It can be either a local asset image (when it is a default image) or an image loaded from a network URL (when is the user's profile image), depending on the *rolepath*. The text content consists of two *Text* widgets. The first displays the team member's role (role). The second *Text* widget displays the team member's name (name). If the name is an empty string, it's displayed as "Nobody".

Sprint Information



(a) Normal User (b) Work User: S.Master, P.Owner/Developer

Figure 6.15: Project Sprint Information when there is no Sprint UI



(a) Sprint not finished (b) Sprint finished

Figure 6.16: Project Sprint Information when there is a Sprint UI

There are two different widgets to be displayed:

1. There is not a sprint for the project. In this case, the *ContainerAddSprint* is displayed.
2. There is a sprint for the project. In this case, the *SprintInfoContainer* is displayed.

6.8.3 *ContainerAddProject* widget

It is utilized in the project, product backlog and the sprint screen when the user does not have any projects and it is a *Container* with the a welcoming message to the user (Hi *username*!), an icon button to add a new project, a text instructing the user to add a project and a Lottie animation.

The add project icon button returns a Dialog to the user. This dialog varies depending on the user role:

- The Scrum master users can only create projects, so they visualize the *DialogCreate*.
- The Product Owners and Developers receive the *DialogJoin*.

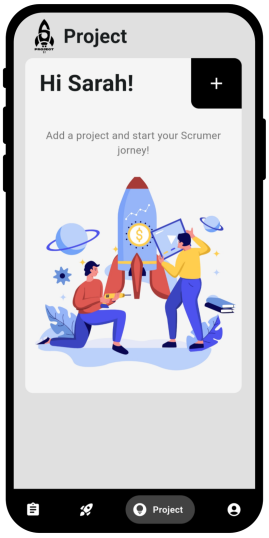
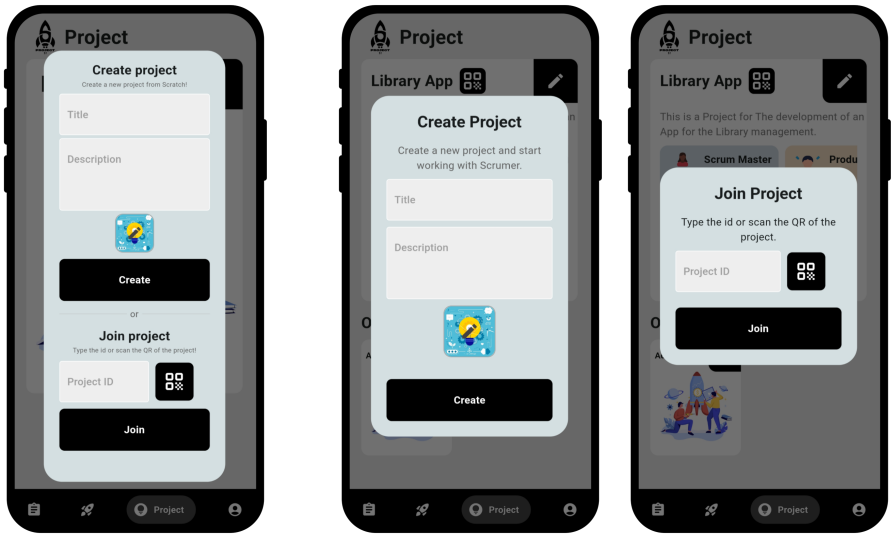


Figure 6.17: Add Project when the user does not have projects UI

- The normal users can create and join projects, so they visualize the *DialogCreate-Join*.



(a) Normal User (b) Work User: S.Master, P.Owner/Developer

Figure 6.18: Dialog to add a Project UI

6.8.4 *DialogCreate* widget

This widget represents a dialog box that allows users to create a new project.

The *DialogCreate* class is defined as a *StatefulWidget* that takes a *UserScrumer* object as a parameter, which is the current user creating the project. It has the variables: two text editing controllers for the title and description of the project. and the *_selectedImage* variable, initially set to null, for the image of the project.

The widget has a *_selectAndCropImage* method that allows the user to select an image from either the gallery or the camera. The selected image is then cropped and stored in the *_selectedImage* variable. For this method, the *ImagePicker* and *ImageCropper* packages are added as a dependency of the project on the *pubspec.yaml* file.[22]

The widget has a *_saveImageToFirebase* method, responsible for saving the project data, including the image, to Firebase Firestore. This method takes place when the user presses the create button.

It first retrieves the values from the title and description text fields. If the title or/and the description are empty, an *AlertDialog* is shown with a message indicating it.

Then, if an image has been selected by the user, it proceeds to upload the image to Firebase Storage. It creates a reference to the storage location using *FirebaseStorage.instance.ref().child(imagePath)*, where *imagePath* is a unique path for the image in the *project_images* directory.

The selected image is uploaded using *putFile(_selectedImage!)*, which returns an *UploadTask* that can be awaited using *await*.

After the image upload is complete, the download URL of the uploaded image is retrieved using *storageRef.getDownloadURL*. To save the download URL into the Firebase Database, a new Project object is created with the project details, including the download URL of the image (if available).

Then, the *joinProject* method is called on the user object (in this case a sub-class: Scrum Master User) to add the project to the user's list of projects and establish it as the current project.

Finally, the dialog is dismissed by calling *Navigator.of(context).pop(true)*.

If no image is selected, the code follows a similar process but without the image upload step. Instead, a default image path is saved for the project.

The build method of the *CreateDialog* constructs the UI of the dialog box using the *AlertDialog* widget. It includes the 'Create Project' title, a message instructing the user to create a new project, a *MyTextField* and *MyTextFieldDescription* (custom widgets from the components folder) for the title and description of the project input fields, an image preview with an editing icon button and a *MyButton* widget used for the "Create"button, which calls the *_saveImageToFirebase* method when tapped.

6.8.5 *DialogJoin* widget

This widget represents a dialog box that allows users to join an existing project.

It has the variables: *idController*, a text editing controller for the project ID input field; *qrController*, a QR code controller used to capture and process QR code scans; and *qrKey*, a global key used to uniquely identify the QR code view widget.

The user can type the Project ID or press the QR Button, which is a *MyButtonIcon* (custom widget from the components folder) for the QR code scan button, which opens a new dialog to display the QR code scanner view.[17] When the user scans a QR code, the *_onQRViewCreated* method is called, which updates the ID controller with the scanned QR code and closes the scanning dialog.

The Dialog has a *MyButton* widget used for the "Join"button. When the user presses it, it first retrieves the project ID from the ID controller and checks if the project exists. If the project does not exist, a SnackBar is shown indicating it.

If the project exists, if the user is already a member of the project, a SnackBar is shown indicating it.

If the user is not a member of the project, it checks if the user has the role of "productOwneränd if there is already a product owner assigned to the project. If so, a SnackBar is shown indicating it.

If the project is in "personal"mode, a SnackBar is shown indicating that the user can't join a project created by a normal user, as this Dialog is only shown to Product Owners and Developers.

If all the checks pass, the user is added to the project using the *joinProject* method of the corresponding user object (sub-class from User): Product Owner or Developer.

The build method of the *JoinDialog* constructs the UI of the dialog box using the *AlertDialog* widget. It includes the 'Join Project' title, a message instructing the user to type or scan the qr to join a Project, a *MyTextField* (custom widget from the components folder) for the ID of the project, the QR Button and the "Join"Button.

6.8.6 *DialogCreateJoin* widget

This dialog is a combination of the Create and the Join Dialog, but it has a differences regarding the sizes of the widgets and on the join section, when the user presses join, the method *joinProject* from the folder *project* is executed.

In the *joinProject* method, it checks if the project does not exist, if the user is already a member of the project, or if the project is in "work"mode (since the *DialogCreateJoin* is shown for normal users). If any of these checks fail, a *SnackBar* is shown to indicate the corresponding error.

If all the checks pass, the user is added to the project using the *joinProject* method of the corresponding user object (sub-class from User): Normal User.

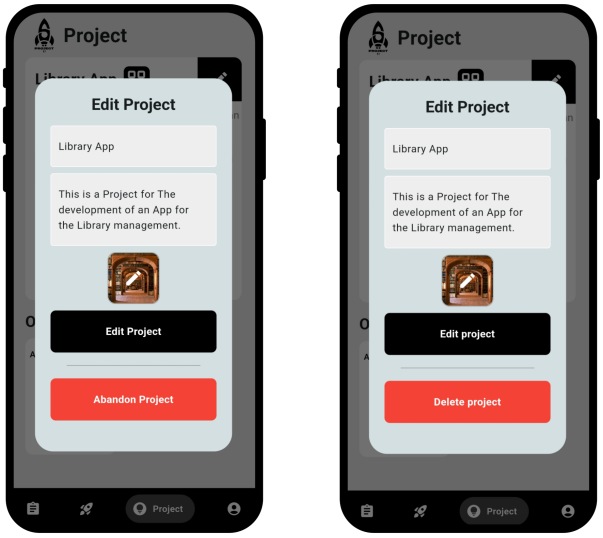
6.8.7 *DialogEditAbandon* widget

This widget is shown when the user presses the *Edit* Button on the *CurrentProjectTile*. This widget is displayed to the Product Owners and the Developers users.

The *DialogEditAbandon* widget is a dialog box that allows the user to edit and abandon a project. It displays the project's current details, such as title, description, and image, and provides options to modify them.

The editing section follows the same process as the create *Dialog* previously described.

For the abandon Project functionality, the widget also includes a Äbandon Project"button that prompts the user to confirm the abandonment of the project, once the Button is pressed. If the user confirms, the project is abandoned calling the *abandonProject* method on the user object.



(a) Abandon Button
-P.Owner
Developer
-Normal user (more members)

(b) Delete Button
-S.Master
-Normal user (the only member)

Figure 6.19: Dialog to edit a Project UI

6.8.8 *DialogEditDelete* widget

This widget is shown when the user presses the *Edit* Button on the *CurrentProjectTile*. This widget is displayed to the Scrum Masters and the Normal users.

The *DialogEditDelete* widget is a dialog box that allows the user to edit and delete a project. It displays the project's current details, such as title, description, and image, and provides options to modify them.

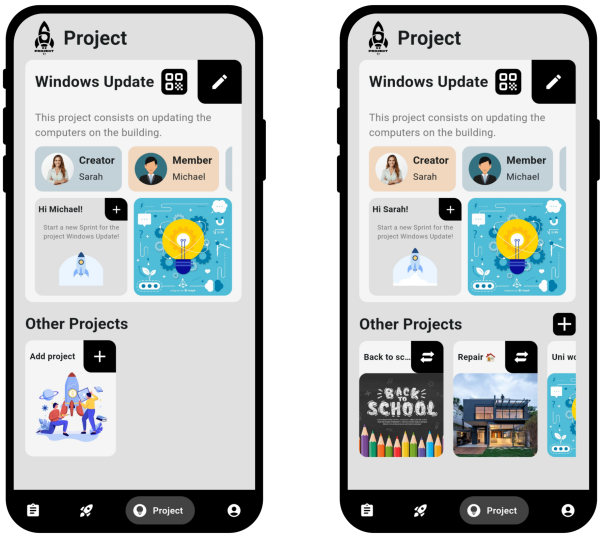
The editing section follows the same process as the create *Dialog* previously described.

For the delete Project functionality, the widget presents a "Delete Project"button that prompts the user to confirm the erasing of the project, once the Button is pressed. If the user confirms, the project is removed from the projects array of the user and from the project is deleted calling the *deletingProject* method on the current project object. Then, the current project takes the null or another project value (if the user has more projects). finally, the user data is updated on the users collection on the database using the *setUserToCollection* method.

For the normal users, if in the project, the normal user is the only member, the delete button is displayed. However, when there are more members in the project, the abandon button, with the same implementation as the Äbandon"button on the *DialogEditAbandon* widget, is displayed.

6.8.9 *OtherProjectTile* widget

The *OtherProjectTile* widget represents a tile for displaying a project other than the current project. It is used in the *ProjectPage* to show the list of other projects that the user is associated with.



(a) One project (b) More projects

Figure 6.20: Other Projects UI

If the user only has one project, a container instructing the user to add a new project is displayed. This container follows the same process as the *ContainerAddProject*.

If the user has more than one project, a container with the project title, a switch button, and the project image are displayed.

If the project title is too long, it is truncated with an ellipsis.

The switch button is represented by an *MyIconButton* (custom widget from the *components* folder). When clicked, it shows an *AlertDialog* confirming if the user wants to

switch to the selected project. When the user confirms, the *currentProject* of the user is set to that project and the database is updated using the method *setUserToCollection*.

The project image is displayed using an Image widget, either from a local asset (default project image) or a network image (image in the fireStore), depending on the project's image path.

With this implementation, all of the use cases and goals regarding the project page have been implemented and achieved.

6.9 Product Backlog

For the implementation of the product backlog section of the Navigation Bar, the sub-folder product backlog includes several widgets. The widget that at the end, is responsible to display the whole project page is the *ProductBacklogPage* widget.

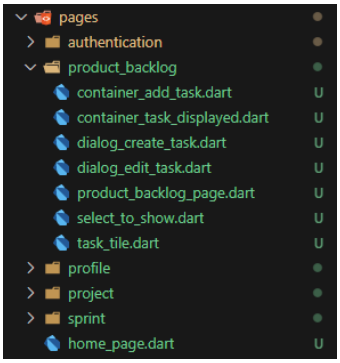


Figure 6.21: *ProductBacklog* sub-folder Structure

6.9.1 *ProductBacklogPage*

This is the widget responsible for displaying all of the UI related to the Product Backlog.

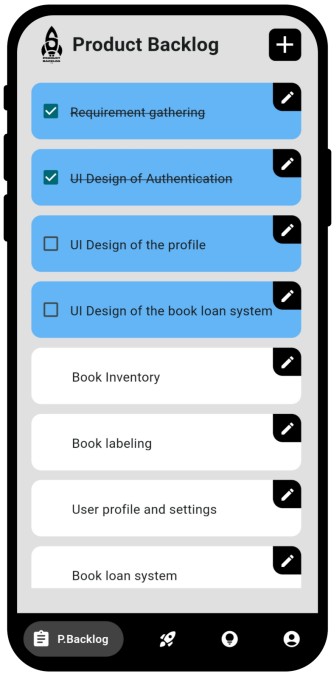


Figure 6.22: Product Backlog UI

Product Backlog UI

The UI consists of a *Scaffold* with various components:

- A Row with the logo of Scrumer, with the modification of having the "PRODUCT BACKLOG"text instead of SCRUMER, and the Product Backlog title.
- If the user has a current project and a current sprint with task; a Add Task Button, is displayed (only to the product owner and the normal user). This Button leads to the *DialogCreateTask*.
- Finally, the method *selectToShow* is used to establish which widget to display.

6.9.2 *DialogCreateTask* widget

This widget represents a dialog box that allows users to create a new task for the current project. This widget is only presented to the product owner and the normal user.

Figure 6.23: Dialog create Task UI

The *DialogCreatetask* class is defined as a *StatefulWidget* that takes a *UserScrumer* object as a parameter, which is the current user creating the project. It has the variables: two text editing controllers for the title and description of the project.

The widget has a *_saveTaskToFirebase* method, responsible for saving the task data. This method takes place when the user presses the create button.

It first retrieves the values from the title and description text fields. If the title or/and the description are empty, an *AlertDialog* is shown with a message indicating it. Then, a new Task object is created with the task details. Finally, the *addTask* method is called on the user object to add the task to the user's current project list of tasks. Finally, the dialog is dismissed by calling *Navigator.of(context).pop(true)*.

The build method of the *DialogCreateTask* constructs the UI of the dialog box using the *AlertDialog* widget. It includes the 'Create Task' title, a message instructing the user to create a new task, a *MyTextField* and *MyTextFieldDescription* (custom widgets from the components folder) for the title and description and a *MyButton* widget used for the "Create"button, which calls the *_saveTaskToFirebase* method when tapped.

6.9.3 *selectToShow* method

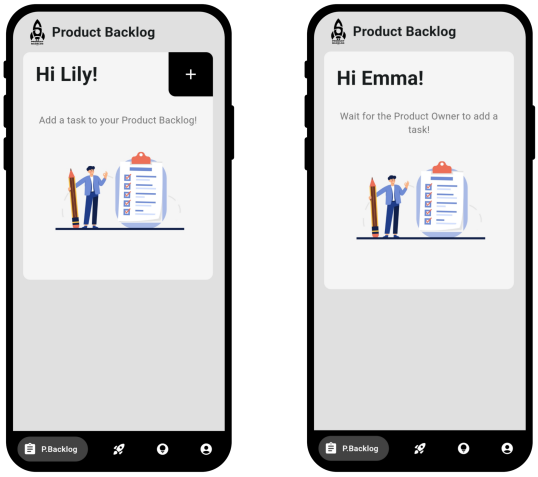
The *selectToShow* function is used to determine which widget to display based on the current state of the user's project and sprint.

1. If the user doesn't have a current project, it returns a *ContainerAddProject* widget, which belongs to the Project folder and it instructs the user to add a new Project.
2. If the user has a current project but doesn't have a current sprint , it returns a *ContainerAddSprint* widget, , which belongs to the Project folder and it instructs the user to add a new Sprint.
3. If the user has a current project, a current sprint, and no tasks in the current sprint, it returns a *ContainerAddTask* widget when the user is in the Product Backlog page, and a *CurrentSprintTile* when the user is on the Sprint Page.
4. If the user has a current project, a current sprint, and there are tasks in the current sprint, it checks if the user is in the sprint page . If it is, it returns a *CurrentSprint-Tile* widget.
5. If none of the above conditions are met, it assumes that the user is in the product backlog page and returns a *ContainerTasks* widget. This widget displays the list of tasks in the product backlog and allows the user to perform related actions.

6.9.4 *ContainerAddTask* widget

It is utilized in the product backlog and the sprint screen when the user does not have any tasks on the current Project and it is a Container with the a welcoming message to the user (Hi *username*!), an icon button, only presented to the product owner and the normal user, to add a new task; a text instructing the user to add a task (product owner and normal user) or to wait for the product owner to add a task (scrum Master and developer); and a Lottie animation.

The add project icon button returns a *DialogCreateTask* to the user.



(a) Add Button
-P.Owner

-Normal user

(b) No Add Button
-S.Master
Developer

Figure 6.24: Add Task when the project does not have tasks UI

6.9.5 *ContainerTasks* widget

It is utilized in the product backlog and the sprint screen when the user has tasks on the current Project and it displays the list of tasks in the product backlog and allows the user to perform related actions.

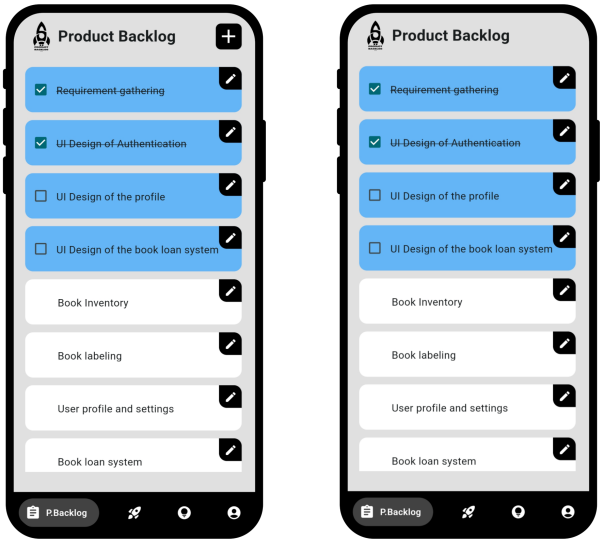
Depending on which page is displayed, its size is different.

It creates a *ListView.builder* widget to build the list of tasks. Inside the *itemBuilder*, for each task, if the user is in the sprint page, the task is displayed only when it is in the sprint.

For each task, it creates a TaskTile widget.

6.9.6 *TaskTile* widget

The TaskTile widget is responsible for displaying the details of each task and providing actions to perform on the task, such as: editing, deleting, and toggling the status of the task. It also allows users to view and add comments to the task.



(a) Add Task
-P.Owner

-Normal user

(b) No Add Task
-S.Master
Developer

Figure 6.25: Tasks displayed on the Product Backlog UI

toggleTaskStatus method

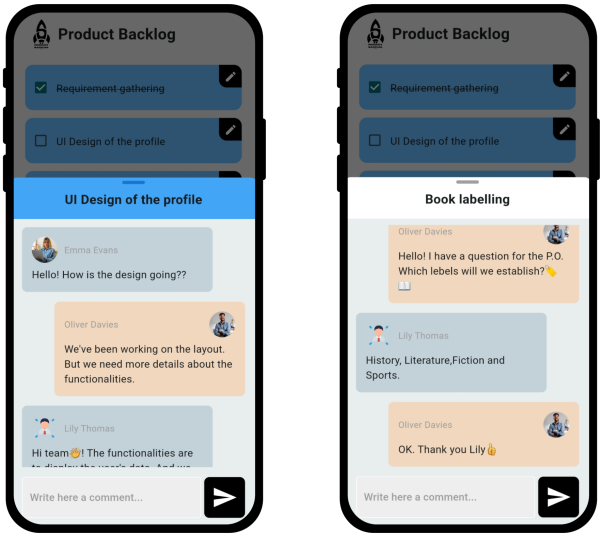
The toggleTaskStatus function is responsible for toggling the status (finished/unfinished) of a task when the checkbox associated with the task is clicked.

showTaskDetails method

The *showTaskDetails* function is responsible for displaying the comments of a task in a modal bottom sheet. It allows users to view the task's title and comments, and add new comments.

Firstly, a commentController is created to handle the input for adding new comments.

The *showModalBottomSheet*[9] function is called to display the modal bottom sheet. The *isScrollControlled* parameter is set to true to enable scrolling when the content exceeds the screen height.



(a) Task in Sprint (b) Task not in Sprint

Figure 6.26: Task comment section UI

The background color of the modal bottom sheet is determined based on whether the task is in a sprint or not. If it's in a sprint, a light grey color is used; otherwise, a darker grey color is used.

The content of the modal bottom sheet is built using a *StatefulBuilder*, which allows the content to be updated within the modal.

The content includes a header section displaying the task's title, a list of comments, and a section for adding new comments.

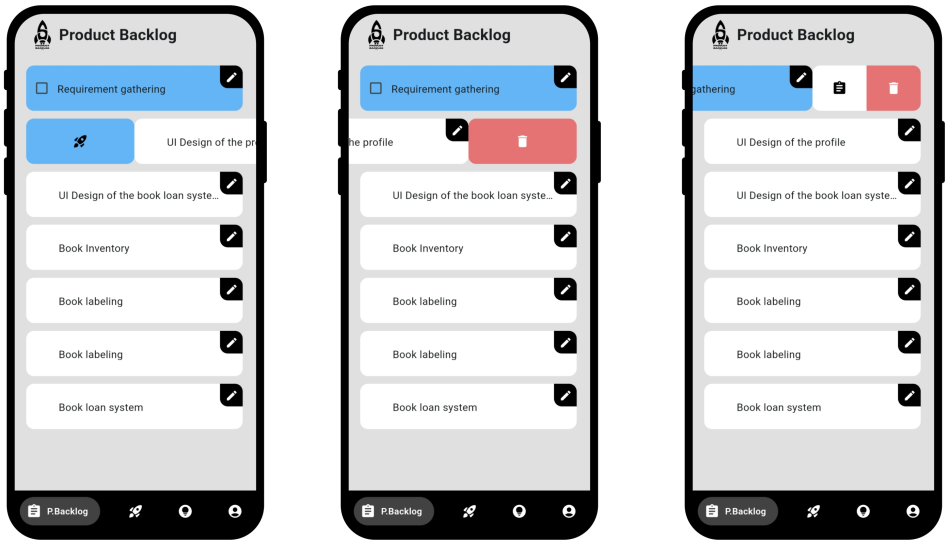
The header section includes a colored bar at the top representing the task's status (blue if in a sprint, white otherwise) with the task's title,

The list of comments is displayed using a *ListView.builder* widget. It iterates over the comments associated with the task and displays them in reverse order. Each comment is styled based on whether the comment was made by the current user or another user.

The section for adding new comments consists of a *MyTextFieldNoPadding* (custom widget from the components folder) for entering the comment text and a *MyButtonIconComment* (custom widget from the components folder) for submitting the comment. When the button is tapped, a new Comment object is created using the input data, and is added tot he task calling the *addComment* method of the *UserScrumer* object.

build method

Inside the build method, it creates an *InkWell* widget that wraps the entire task tile. This allows users to tap on the tile to execute the *showTaskDetails* method and access the comments of the task.



(a) Add Task to Sprint (b) Delete Task from P.Backlog (c) Delete Task from Sprint

Figure 6.27: Task actions UI

The task tile is wrapped in a *Slidable* widget[25], which provides swipe actions for the task tile. Depending on the context, different actions are available:

- If the user is in the product backlog or the sprint page and the task is in the Sprint, the user can swipe left and press on the white action with a assignment icon to delete the task from the sprint. This will execute the *deleteTaskFromSprint* method on the user object.
- If the user is in the product backlog page, the user can swipe left and press on the red action with a bin icon to delete the task. This will execute the *deleteTask* method on the user object.

- If the task is not the sprint, the user can swipe right and press on the blue action with a rocket icon to add the task to the sprint. This will execute the *addTaskToSprint* method on the user object.

The task tile has a background color that indicates whether the task is in the sprint (blue) or not (white).

The content of the task tile includes:

- A *Checkbox* (if the task is in the sprint) which triggers the *toggleTaskStatus* method when pressed, the task title.
- The title with different styles depending on whether the task is marked as finished (line through the text) is or not.
- An edit button which displays the *DialogEditTask* when pressed.

6.9.7 *DialogEditTask* widget

This widget represents a dialog box that allows users to edit a task for the current project.

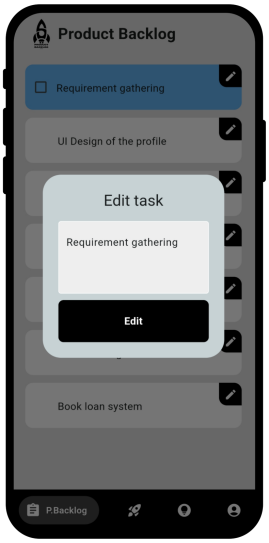


Figure 6.28: Dialog edit Task UI

The *DialogEditTask* class follows the same process as the *DialogCreateTask* widget, except for the fact that when the Ëdit"button is pressed, the *editTask* method is called on the user object to edit the task to the user's current project list of tasks.

With this implementation, all of the use cases and goals regarding the product backlog page have been implemented and achieved.

6.10 Sprint

For the implementation of the sprint section of the Navigation Bar, the sub-folder sprint includes several widgets. The widget that at the end, is responsible to display the whole sprint page is the *SprintPage* widget.

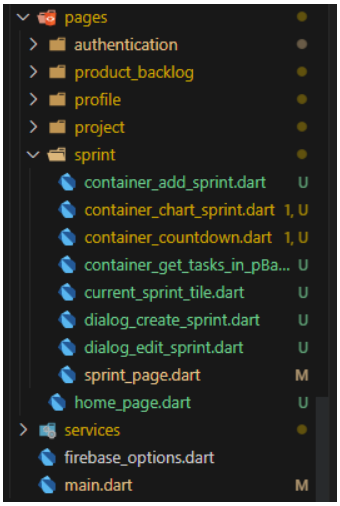


Figure 6.29: *Sprint* sub-folder Structure

6.10.1 *SprintPage*

This is the widget responsible for displaying all of the UI related to the Sprint.

Sprint UI

The UI consists on a *Scaffold* with various components:

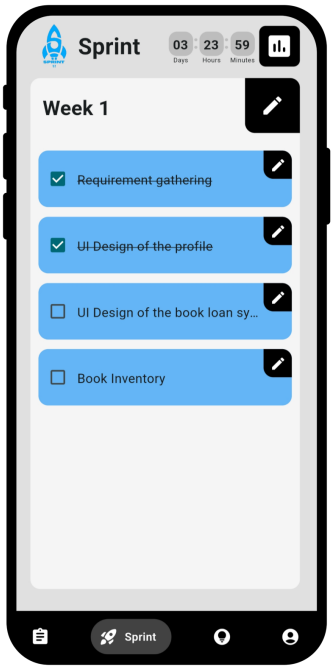


Figure 6.30: Sprint UI

- A Row with the logo of Scrumer, with the modification of having the SSPRINT"text instead of SSCRUMER", and the Sprint title. If there is a current sprint and it is not finished, the logo is blue.
- If the user has a current project and a current sprint, a *CountdownTimerContainer* widget is displayed.
- Finally, the method *selectToShow* (explained in the product Backlog section) is used to establish which widget to display.

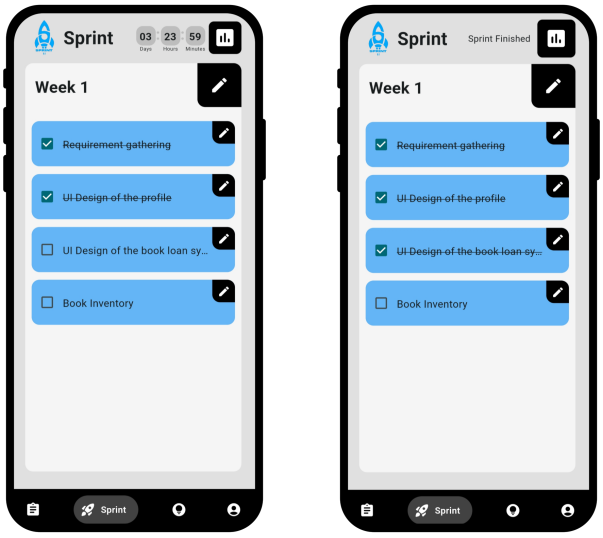
6.10.2 *CountdownTimerContainer* widget

This widget is responsible for displaying the time left of the current sprint and the chart Button (which displays the Sprint report). It is used in the Sprint page and in the Project page (to display the sprint information). In each page, the sizes vary, based on the value of *onReport*, that indicates whether the countdown is on the project page.

It uses the *CountdownTimer* widget from the *flutter_countdown_timer*[11] package (dependency established on the *pubspec.yaml* file).

The *CountdownTimer* widget takes the *endTime* parameter, which represents the end time of the countdown in milliseconds since the epoch. We pass the *end* attribute of the current sprint object.

The *widgetBuilder* callback is invoked by the *CountdownTimer* widget to build the UI based on the current remaining time.



(a) Sprint not finished (b) Sprint finished

Figure 6.31: Sprint Countdown UI

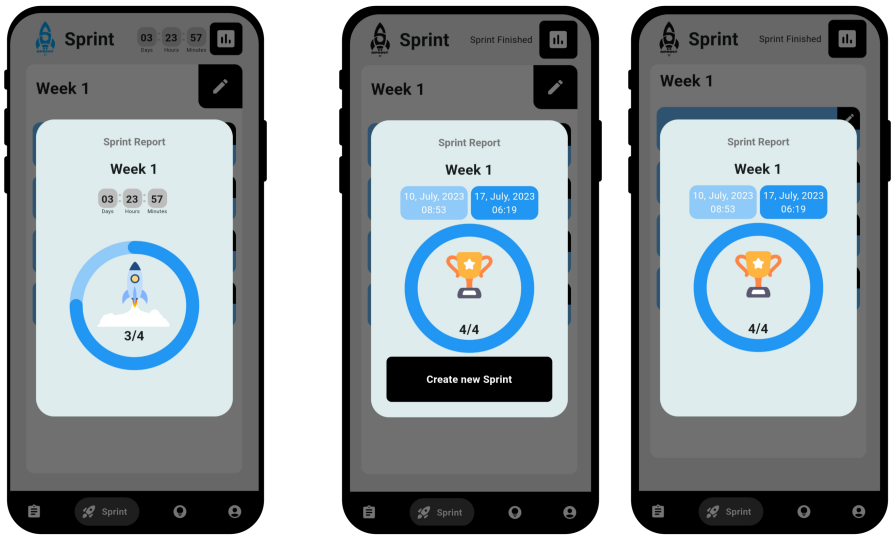
If the sprint is not finished (time is null), indicating that the countdown has finished, it displays a message stating that the sprint has finished.

If the sprint is not finished, it builds a row with three *_buildTimeContainer* widgets to display the remaining days, hours, and minutes.

If the user is on the sprint page, it also shows a Chart button with an icon to open a dialog for the sprint report, which builds the *SprintInfoContainer* widget.

6.10.3 *SprintInfoContainer* widget

This widget displays information about a sprint, including its title, start and end dates, progress percentage, and optional buttons for creating a new sprint. It is used in the Sprint page, in the *CountdownTimerContainer* when the Chart Button is pressed. Also, it is used in the project page, to display the sprint information.



(a) Sprint not finished

(b) Sprint finished:
1. S.Master/Normal, 2. P.Owner/Developer

Figure 6.32: Sprint Report UI

It first displays the sprint title. Secondly, when the user is in the sprint page, there are two possibilities:

- If the sprint is finished, it displays the start and end dates of the sprint in a horizontal row.
- If the sprint is not finished, it displays the *CountdownTimerContainer* widget to show the time left for the sprint.

It uses the *CircularPercentIndicator* widget from the *percent_indicator* package (dependency established on the *pubspec.yaml* file) to display the progress percentage of the sprint.

Inside the *CircularPercentIndicator*[28], it displays an animated Lottie animation and the number of finished tasks and total tasks in the sprint. The animation indicates either that the sprint is finished (trophie animation) or it is not (rocket launching animation).

If the sprint is over and the user's role is either "normalör ßcrumMaster", it displays a button for creating a new sprint, opening the *DialogCreateSprint* widget.

6.10.4 *selectToShow* method

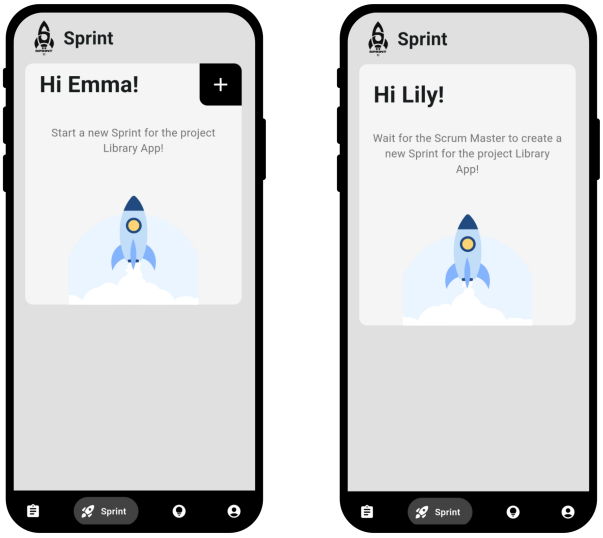
The method *selectToShow* (explained in the product Backlog section) is used to establish which widget to display:

- If the user doesn't have a current project, it returns a *ContainerAddProject* widget, which belongs to the Project folder and it instructs the user to add a new Project.
- If the user has a current project but doesn't have a current sprint , it returns a *ContainerAddSprint* widget.
- If the user has a current project, a current sprint, it returns a *CurrentSprintTile* widget.

6.10.5 *ContainerAddSprint* widget

It is utilized in the product backlog and the sprint screen when the user does not have a sprint fro the current project and it is a Container with the a welcoming message to the user (Hi *username*!), an icon button to add a new sprint (only displayed to the scrum aster and the normal user), a text instructing the user to add a project (scrum master and normal user) or to wait for the scrum master to create a sprint (product owner and developer) and a Lottie animation.

The add project icon button returns a Dialog to the user, the *DialogCreateSprint*.



(a) Add sprint
-S.Master

-Normal user

(b) No Add Sprint
-P.Owner
Developer

Figure 6.33: The project does not have a sprint UI

6.10.6 *DialogCreateSprint* widget

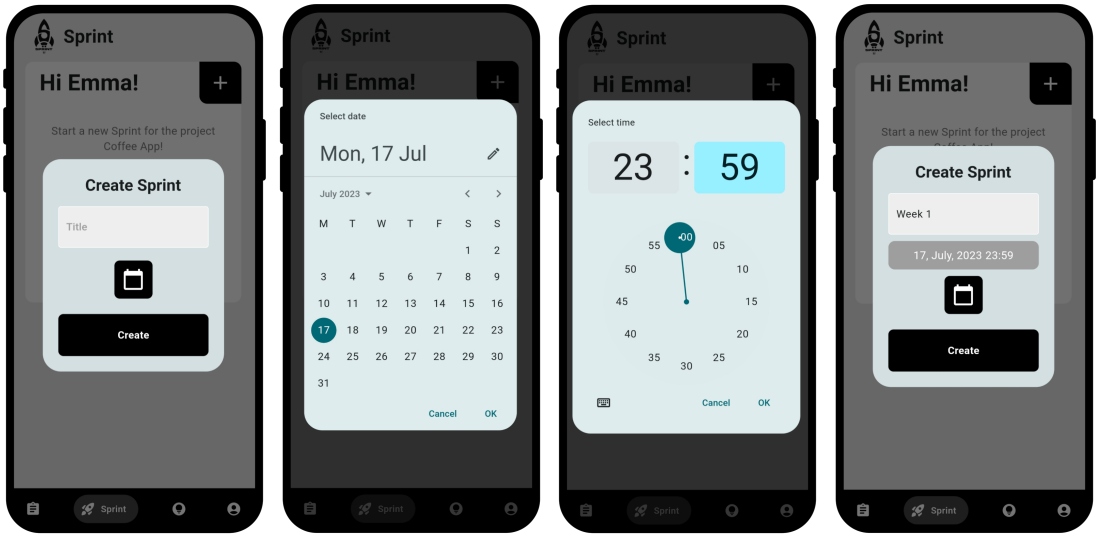
This widget represents a dialog box that allows users to create a new sprint for the current project. This widget is only presented to the scrum master and the normal user.

The *DialogCreateSprint* class is defined as a *StatefulWidget* that takes a *UserScrumer* object as a parameter, which is the current user creating the sprint. It has the variables: a text editing controller for the title of the sprint , a *selectedDateTime*, a *formattedDate* and a *formattedTime* variables.

The widget has a *_showDateTimePicker* method, responsible for showing a date picker and a time picker and allowing the user to select a future date and time for the sprint. [33]

Localization configuration

To be able to have the calendar display where the first day of the week is monday [23], the package *flutter_localizations* is added as a dependency and we configure *localization* for



(a) Create Sprint (b) Date Picker (c) Time Picker (d) Selected Date

Figure 6.34: Dialog create Sprint UI

our App with a list of delegates (*GlobalMaterialLocalizations.delegate*, *GlobalWidgetsLocalizations.delegate*, *GlobalCupertinoLocalizations.delegate*), that handle the localization of various widgets; and a list of language and region (*English*, *Spanish*); and finally, we add the *locale* property to the date picker as Great Britain.

In this method, it first uses the *showDatePicker* function to display a date picker dialog.

If a date is selected, it shows the time picker dialog using the *showTimePicker* function.

If a time is selected, it creates a *DateTime* object called *selectedDateTime* by combining the selected date and time.[16]

Then, if the selected date and time is before the current date and time, an *AlertDialog* with an error message is shown.

If the selected date and time is valid, it updates the *selectedDateTime*, *formattedDate*, and *formattedTime* variables.

The widget has a *_saveTaskToFirebase* method, responsible for saving the sprint data. This method takes place when the user presses the create button.

It first retrieves the value from the title text field. If the title or/and the *selectedDateTime* are empty, an *AlertDialog* is shown with a message indicating it. Then, a new Sprint

object is created with the sprint details, giving the value of *Datetime.now()* to the begin attribute and the *selectedDateTime* to the end attribute. Finally, the *addSprint* method is called on the user object to add the sprint to the user's current project list of sprints and as the currentSprint.

The build method of the *DialogCreateSprint* constructs the UI of the dialog box using the *AlertDialog* widget. It includes the 'Create Sprint' title, a *MyTextField* (custom widget from the components folder) for the title, a *MyIconButton* widget for selecting the end date and time of the sprint, which calls the *_showDateTimePicker* method when tapped; and a *MyButton* widget used for the "Create"button, which calls the *_saveSprintToFirebase* method when tapped.

6.10.7 *CurrentSprintTile* widget

The *CurrentSprintTile* widget displays the details of the current sprint of the current project for a user. It includes the sprint's title and an edit button, for the scrum master and the normal user, which displays the *dialogEditSprint* widget.

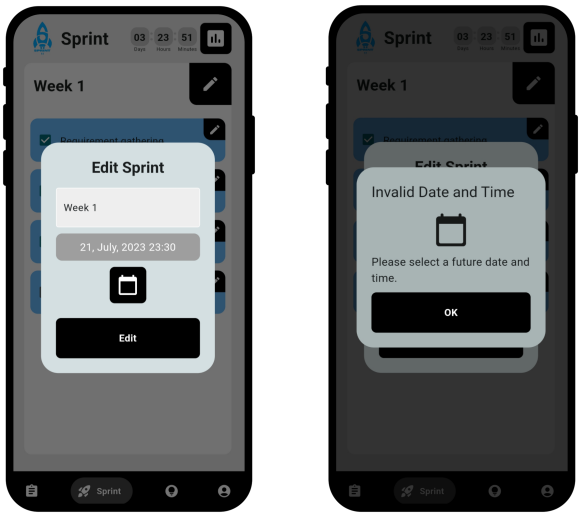
There are two possible cases:

1. The sprint does not have tasks. In this case, the *ContainerGetTasksInPBacklog* is return.
2. The sprint has tasks. In this case, a *ContainerTasks* is return.

6.10.8 *DialogEditSprint* widget

This widget represents a dialog box that allows users to edit the sprint title and end date. It is only displayed to the scrum master and normal user.

The *DialogEditSprint* class follows the same process as the *DialogCreateSprint* widget, except for the fact that when the Ëdit"button is pressed, the *editSprint* method is called on the user object to edit the sprint.



(a) Edit Sprint (b) Date error

Figure 6.35: Dialog edit Sprint UI

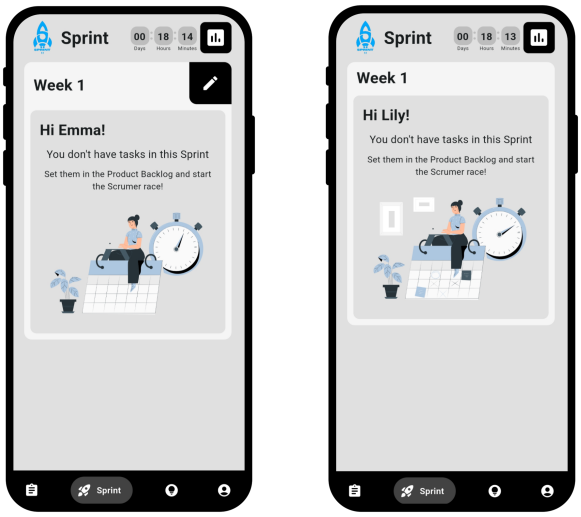
6.10.9 *ContainerGetTasksInPBacklog* widget

This widget is responsible for displaying a message when a user doesn't have any tasks in the current sprint.It consists on a *Column* with a welcoming message to the user (Hi *username*!), a text informing the user that he/she doesn't have any tasks in the current sprint,a text instructing the user to set tasks in the product backlog; and a Lottie animation widget.

6.10.10 *ContainerTasks* widget

It is utilized in the product backlog and the sprint screen when the user has tasks on the current Project and in the sprint page, it displays the list of tasks in the sprint and allows the user to perform related actions.

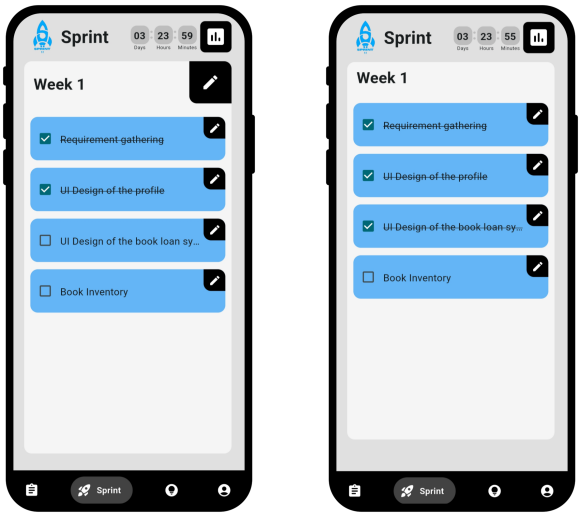
It creates a *ListView.builder* widget to build the list of tasks. Inside the *itemBuilder*, for each task, if the user is in the sprint page, the task is displayed only when it is in the sprint. As it was explained in the product backlog section, for each task, it creates a *TaskTile* widget, which allows the user to mark or unmark the task as done, edit the task, access the comment section, and use the *Slidable* feature to delete the task from the sprint or delete it from the project.



(a) Edit sprint
-S.Master
-Normal user

(b) No Edit
-P.Owner
Developer

Figure 6.36: Sprint without tasks UI



(a) Edit Button
-S.Master

-Normal user

(b) No Edit Button
-P.Owner
Developer

Figure 6.37: Sprint with tasks UI

With this implementation, all of the use cases and goals regarding the sprint page have been implemented and achieved.

6.11 Profile

For the implementation of the profile section of the Navigation Bar, the sub-folder profile includes several widgets. The widget that at the end, is responsible to display the whole profile page is the *ProfilePage* widget.

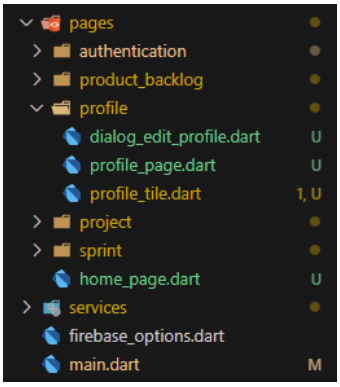


Figure 6.38: *Profile* sub-folder Structure

6.11.1 *ProfilePage*

This is the widget responsible for displaying all of the UI related to the Profile. The UI consists on a *Scaffold* with various components:

- A *Row* with the logo of Scrumer, with the modification of having the "PROFILE"text instead of SSCRUMER", and the Sprint title.
- A logout Button that executes the *signUserOut* method, which uses

  FirebaseAuth.instance.signOut() to log the user out.
- A *ProfileTile* widget, which displays the user's information.

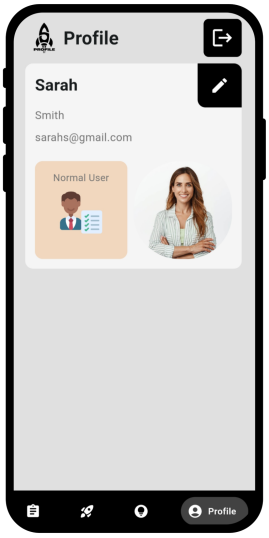
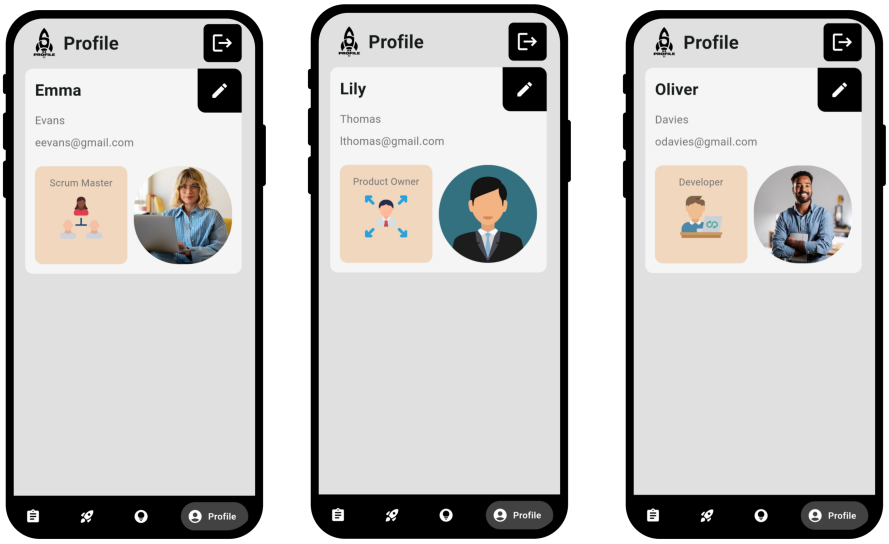


Figure 6.39: Normal User Profile UI

ProfileTile widget

This widget is responsible for displaying the profile information of a user.



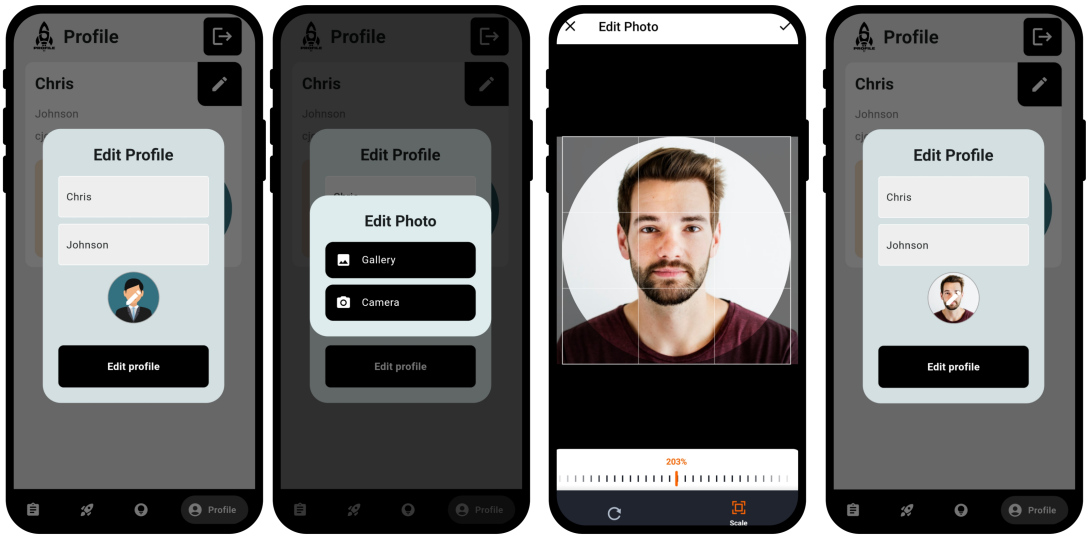
(a) Scrum Master (b) Product Owner (c) Developer

Figure 6.40: Profile UI

It consists on a *Column* with the following elements: a row with the user's name (if the name overflows, it is truncated with an ellipsis) and an edit button, which opends a *DialogEditProfile* widget; the user's surname; the user's email and a row with the the role information and profile image.

DialogEditSprint widget

This widget represents a dialog box that allows users to edit the user's name, surname and profile photo.



(a) Default Image (b) Image Picker (c) Image Cropper (d) Image selected

Figure 6.41: Edit Profile UI

The widget has a *_selectAndCropImage*[22] method that allows the user to select an image from either the gallery or the camera.

The widget has a *_saveImageToFirebase* method, responsible for saving the project data, including the image, to Firebase Firestore. This method takes place when the user presses the edit button.

It first validates that the name and surname fields are not empty. Otherwise, the profile image is saved to Firebase Storage. If a new image was selected, the image is uploaded

to Firebase Storage and the download URL is obtained. The user's profile information is then updated with the new *name*, *surname*, and *imagePath*.

If the user had a previous profile image that was not the default image, it is deleted from Firebase Storage.

Finally, the user information is updated in Firebase Firestore.

With this implementation, all of the use cases and goals regarding the profile page have been implemented and achieved.

6.12 Logo

To add the logo to the App icon for launching the app, we first use the tool *appicon*, to have the icon in all necessary sizes for the platforms *ios* and *android*. As a result, we obtain two different folders, one for each operating system.

Secondly, we add the generated icons to the corresponding android (*android/app/src/-main/res*) and ios (*ios/Runner/assets.xcassets/appIcon.appiconset*) folders.[12]

7 Evaluation

In this chapter, we include the testing process for Scrumer. First, we introduce the testing strategy; then, we explain the unitary tests and the manual tests. Finally, we explain the acceptance tests.

7.1 Strategy

Observing the requirements established and the implementation of the App, the strategy chosen is to use unitary tests to evaluate the methods used on the models of the App, and manually test (System tests) the App to evaluate the User Interface.

This strategy is chosen because it is easier to check all of the widgets (sometimes it ends up being a big widget tree) displayed for each case directly on the User Interface; rather than verifying each widget for each case, which has a lot of possibilities regarding the 4 types of users, number of projects, sprint finished or not, etc.

Therefore, the unitary tests are aimed to evaluate of the changes on the class models are done correctly, which means testing the methods of these classes. And the manual testing includes evaluation of the whole system, including the UI that calls these methods; therefore, testing the integration between widgets and the database changes shown in the UI.

The manual testing can be structure as functional tests and non functional tests, corresponding to the different requirements previously established. We use real Android devices: Samsung Galaxy A14, Samsung Galaxy Core Prime and Samsung Galaxy A50.

Finally, acceptance testing takes place, showing the product to four different users.

7.2 Unitary Tests

In Flutter, testing in the actual firebase project is not available. Therefore, we use the package *mockito*, to imitate a Firebase Authetication Database, a Firebase Firestore Database and a Firebase Cloud Storage.

Then, in the test folder, we include a file *widget_file.dart*, where the tests are included. In this file, a test for the display of all the element of a UI widget is included, to serve as an example for other tests. The widget tested is *DialogCreate*, used for creating a project.

```
testWidgets('Test Create Project Dialog Error Message',
      (WidgetTester tester) async {
    // Create a test user
    String id = '01';
    ScrumMasterUserScrumer user = ScrumMasterUserScrumer(
      id: id,
      name: 'philip',
      surname: 'smith',
      email: "philip@gmail.com",
    );
    await tester.pumpWidget(
      MaterialApp(
        home: DialogCreate(
          user: user,
          firebaseAuth: mockFirebaseAuth,
          firebaseFirestore: mockFirebaseFirestore,
        ),
      ),
    );
    // Verify that the dialog appears
    expect(find.text('Create Project'), findsOneWidget);
    expect(find.text('Title'), findsOneWidget);
    expect(find.text('Description'), findsOneWidget);
    // Tap the "Create" button
    await tester.tap(find.byType(MyButton));
    await tester.pump();
    // Verify that the title and description are needed
    expect(find.text('Title needed'), findsOneWidget);});
```

Code snippet 7.1: Test method for the error message in the Create Project Dialog

We test if a widget containing an error message is displayed when the user does not enters any project title and presses the "Create"button.

The rest of the tests follow a similar structure and evaluate if the methods used for different features like edit profile user, add multiple projects to one user, create a project as Scrum Master and later, add a product owner to the same project, add a sprint to a project.

The goal of the unitary tests is to find errors in each model, the file is executed and the tests have a successful result.

7.3 Functional Evaluation

This is the evaluation corresponding to each functional requirements of the App Scrumer.

7.3.1 Functional Requirements for Users (FR01 to FR06)

FR01: A user has to be able to register and login using an email and password.

Evaluation: We test the user registration process, ensuring that the user can input their name, surname, and email in both registration methods. Upon successful registration, we verify that the user is redirected to the home page, and their information is correctly stored in the Authentication Database and Firestore Database.

FR02: A user has to be able to login using an email and password used on the registration.

Evaluation: We check the login process, making sure that users can log in using their registered email and password. In case of incorrect credentials, we verify that an appropriate error message is displayed, and upon successful login, the user is directed to the home page.

Regarding the four type of users, we observe that four sub-classes have been established in the app models.

Furthermore, regarding the two different registrations, we can see that, first, a interface to choose the registration is built followed by two different registration interfaces for the two different users.

FR03: A user has to be able to reset his/her password.

Evaluation: We test the password reset functionality, confirming that when a user requests a password reset, they receive an email to reset their password. After following the reset link, we check if the user can successfully reset their password.

FR04: A user has to be able to register and login using a Google account.

Evaluation: We verify that the user can choose the Google account registration option and successfully register using their Google account. Similarly, we test if the user can log in using their registered Google account. Upon successful registration and login, we ensure that the user is directed to the home page and their credentials are appropriately stored in the Authentication Database and Firestore Database.

FR05: A user can have multiple projects.

Evaluation: We create multiple projects for a user and verify that all projects are correctly associated with the user in the database. (The method to add multiple projects is tested in the unitary tests). Additionally, we check the user interface to ensure that all projects are displayed correctly for the user.

FR06: A user has the fields Name, Surname, and Photo, which can be modified.

Evaluation: We test the user profile editing functionality, ensuring that the user can modify their Name, Surname, and Photo. After making the changes, we verify that the updated information is correctly stored in the database and reflected in the user interface.

FR07: A project has the fields Name, Surname, and Photo, which can be modified.

Evaluation: We test the user profile editing functionality, ensuring that the user can modify their Name, Surname, and Photo. After making the changes, we verify that the updated information is correctly stored in the database and reflected in the user interface.

7.3.2 Functional Requirements for Projects (FR07 to FR08)

FR07: A project has the fields Title, Description, and Photo, which can be modified.

Evaluation: We verify that a project's Title, Description, and Photo can be modified through the user interface. After making the changes, we check if the updated project information is correctly stored in the database and reflected in the user interface.

FR08: A project has tasks and a current Sprint, which can also be modified.

Evaluation: We test the functionality to modify tasks and the current Sprint of a project. We verify that tasks can be added, removed, and edited through the user interface, and the changes are correctly reflected in the database. Additionally, we ensure that the current Sprint can be modified, including its Title, Starting Date, and Finish Date.

7.3.3 Functional Requirements for Projects (FR09 to FR17)

FR09: A normal user can create Projects and Join Projects created by other normal users.

Evaluation: We test the ability of a normal user to create a new project. Upon successful creation, we check if the project is added to the user's project list.

Additionally, we test the ability of a normal user to join a project created by another normal user using a generated QR code. Upon successful joining, we verify if the user is

added to the project's user list and if his/her current project and other projects UI has been updated. Then, we verify if the new member is visible for the rest of the project members.

We also test possible errors, by entering a project Id that does not exist, the result is a message telling us that the project does not exist. Also, we test it entering a "personal"project and the result is a message telling us that we need to join "work"projects. Finally, we test that a user can join a project for which he/she is already a member. The result is a message telling that he/she is already a member.

FR10: A normal user can abandon a project, and when there's no one else on the project, delete the project.

Evaluation: We test if a normal user can abandon a project, this is observe in the Dialog widget that is displayed with a Äbandon Project". If the user is the last member of the project, we check if the button appearing is the "Delete Button". Then, we evaluate if the corresponding changes in the users and project data take place. For the delete option, we observe that the project document is erase from the Firebase database.

FR11: A scrum Master can create Projects.

Evaluation: We test the ability of a Scrum Master to create a new project and ensure it is successfully added to the database.

FR12: A product owner and a developer can join projects.

Evaluation: We verify that both product owners and developers can only join existing projects using the provided QR code. We do this by we entering a project Id that does not exist, the result is a message telling us that the project does not exist. Also, we test it entering a "personal"project and the result is a message telling us that we need to join "work"projects.

Finally, we test that a product owner or a developer can join a project for which he/she is already a member. The result is a message telling that he/she is already a member.

FR13: The method to join projects needs to be the following: A QR for the project needs to be generated and displayed to be read by another user who uses it to join the project.

Evaluation: We check if the QR code generation is successful and whether it can be scanned by another user to join the project.

FR14: A project needs to have a mode field, which is "work"when the project has been created by a Scrum Master and "normal"when the creator was a normal user.

Evaluation: We verify if the mode field on the firebase Database of the project is set correctly based on the creator's role.

FR15: A "work"project needs to have one Scrum Master and it can have zero or one Product Owner. There can be multiple developers.

Evaluation: We check if the project enforces the correct number of Scrum Masters, Product Owners, and Developers based on the project mode. To test it, we try to join as Product Owner a project, where there is already another product owner. The result is a message saying that there is another product owner for that project.

For developers, we test if for one project, multiple developers can join.

FR16: A product Owner and a developer can abandon a project.

Evaluation: We test if a product owner or developer can abandon the project and verify that they are removed as members of the project by looking in the database and in the Project UI of other members.

FR17: A scrum Master can delete a project.

Evaluation: We ensure that a Scrum Master can delete a project and that the associated document is also removed from the database. Also, we verify that the project has also been erase for all project members.

7.3.4 Functional Requirements for Current Sprint (FR18 to FR21)

FR18: A sprint has the fields Title, Starting Date, and Finish Date, which can be modified.

Evaluation: We test if the fields of the current sprint, such as Title, Starting Date, and Finish Date, can be modified successfully.

FR19: A normal user can create a Sprint and edit its duration.

Evaluation: We verify if a normal user can create a new sprint and edit its duration (Starting Date and Finish Date).

FR20: A scrum Master can create a Sprint and edit its duration.

Evaluation: We test if a Scrum Master can create a new sprint and modify its duration. We verify that a product owner and a developer cannot create or modify sprints by observing that the edit button does only appear for the scrum master.

FR21: All users need to see the progress of the sprint, meaning the tasks completed.

Evaluation: We check if all users can view the progress of the sprint, by observing the sprint report.

7.3.5 Functional Requirements for Tasks (FR22 to FR24)

FR22: A task has the field Title which can be modified.

Evaluation: We verify if the Title field of a task can be modified successfully.

FR23: A task can be created by a normal or a product owner user.

Evaluation: We test if both normal users and product owners can create new tasks. We observe that the other users cannot create tasks, as the create button does not appear for them.

FR24: A task can be added to or removed from the sprint and deleted from the project.

Evaluation: We check if tasks can be added to or removed from the current sprint and if they can be deleted from the project by using the displayable options. We check if the data has been successfully modified in the database.

FR25: A task has a comment section, which acts as a chat for users on the project to collaborate.

Evaluation: We examine the functionality of the comment section associated with tasks in the Scrumer app. We ensure that each task in the app has a comment section associated with it. When viewing a task, we ensure that the user views the list of comment, being the last comment on the bottom part.

We ensure that users can type and add comments to the comment section of a task. Then, we test whether the comments added are updated in the database.

Finally, we evaluate the user Identification: each comment should display the name, surname and image of the user who posted it.

7.4 Non-Functional Evaluation

In this section, we evaluate the non-functional requirements for Scrumer:

NFR01: The App needs to be compatible with both Android and iOS Operating Systems.

Evaluation: We verify if the app functions correctly on both Android, using several real Android Devices and we verify if all of the IOS configurations adhere to the flutter requirements for IOS, as having developed the app in Windows doesn't allow testing the app in real iPhones.

NFR02: Every screen or component needs to be charged in less than 5 seconds with the aim of not losing the user's attention.

Evaluation: We measure the loading times of various screens and components to ensure they meet the specified requirement.

NFR03: The error messages have to be clear and easy to understand by the user.

Evaluation: We review the error messages displayed to the user and ensure they are clear and provide relevant information.

NFR04: The design of the App needs to have a uniform theme, as well as intuitive user interfaces.

Evaluation: We assess the app's design to ensure consistency in the theme, which we observe follows a light grey design, with the use of black and blue (sprint related actions) colors, and evaluate if the user interfaces are intuitiveness.

NFR05: Responsive design: the app needs to be adjustable to different screen sizes.

Evaluation: We test the app on various Android Devices with different screen sizes and use different orientations in those devices.

7.5 Acceptance Testing

Acceptance testing takes place to show the product to four different users. This step is crucial to gather feedback from actual users and validate that the app meets their requirements and expectations.

During acceptance testing, the four users interact with the app, perform various tasks, and provide feedback on their experience.

During the acceptance testing phase, the first user, who uses the app as a normal user, provides feedback indicating that initially, he found it difficult to understand how to add tasks and was confused by the concept of sprints. However, once he pressed on the Add button in the Container to Add a Sprint, the calendar and timer helped him to understand the time slot system and found the app useful for his daily tasks. To address this usability concern, descriptions have been added under the "Create Sprintänd Ädd Task"features to provide clearer guidance and improve user understanding.

The other three users use the app as a Scrum Team, and their feedback is positive, particularly highlighting the benefits of collaboration through the task comments feature. The Scrum Master appreciates the ease of creating projects and allowing others to join, as it enhanced productivity and team collaboration. However, they expressed their concern of having everyone (despite the role) with the same profile picture, when they have the default profile picture.

In response to their feedback, a new feature has been added to the app that when the user does not have a personalized photo, it does not display the default user photo, but a different profile picture for each role in the members section of the current project and in the comment section. This addition helps team members easily identify one another during discussions.

Once the acceptance testing phase is completed, the app is ready for its official release to the target audience.

Part III

Epilogue

8 Conclusion

Having finish with the development of Scrumer, this chapter includes the conclusions, including an analysis of the goals achieved and the lessons learned from the project. Furthermore, it is important to analyse the improvement that could be made to the software and the future steps to take.

8.1 Goals achieved

The main goal of this project was to develop a functional mobile application, Scrumer, that successfully implemented the design and requirements established for supporting agile project management using the SCRUM framework. Several goals were achieved throughout the process:

- Understanding the Context: A thorough analysis of existing project management apps, both related and unrelated to Scrum, was conducted. This analysis provided valuable insights into the market and helped in identifying the unique features and aspects that could be incorporated into Scrumer.
- Clear Requirements Definition: The functional and non-functional requirements of Scrumer were defined, covering essential features such as user registration and login, project creation and management, task creation and collaboration, and sprint planning. These requirements provided a solid foundation for the development process.
- Use Case Analysis: The use cases corresponding to the defined requirements were analyzed and documented. This analysis played a crucial role in shaping the design and implementation of Scrumer, ensuring that the app's functionality aligned with the identified use cases.

- System Architecture and User Interface Design: A clear and comprehensive design of the system architecture and user interface was established. This design served as a roadmap for the implementation phase and ensured that the app met the intended requirements and provided an intuitive user experience.
- Adjusted Implementation to design and requirements: a working App was built, using a Flutter building environment and Firebase for the authentication database, App database and Image Cloud Storage.
- Evaluation of the App, an analysis was made, to ensure that each requirement was successfully implemented and met the intended functionality.

Regarding the impact of our App, Scrumer improves productivity of Task Management within a Scrum cycle by focusing on four concepts:

- User Productivity Enhancement: Scrumer successfully enhanced task management productivity within Scrum cycles by providing a mobile platform for users to access and update their tasks conveniently. The app's user-friendly interface and role-based access control contributed to improved efficiency and streamlined collaboration within Scrum teams.
- Flexibility for User Scrum Roles: Providing flexibility in accommodating the user roles within projects: Scrum Masters, Product Owners, and Developers; allowed for better customization and adaptability to specific project needs. Understanding the unique requirements of each role and implementing features accordingly contributed to a more tailored user experience.
- Flexibility for Non-Scrum Users: Scrumer not only aimed to Scrum teams but also introduced individuals to the Scrum framework by offering a flexible mode for personal task management. This flexibility expanded the app's user base and allowed users to benefit from Scrum principles even outside traditional project management settings.
- Streamlined Project Management: The integration of multiple projects within Scrumer, particularly for Scrum Masters working in companies with a Scaled Agile Framework like LeSS, provided an efficient and centralized platform for managing various projects. The ability to switch between projects seamlessly and initiate team projects without the need for extensive setup further enhanced project management processes.

8.2 Lessons learned

Throughout the development of Scrumer, several valuable lessons were learned:

- Simplicity in Design: Simplifying the class structure and database design proved to be crucial in managing different user types and roles within projects. For example, the fact that the projects were stored in the same database and were the same for both types of users, made it difficult to differentiate when to display the team members on the project page as work users (scrum master, product owner, developers) or as normal users (creator, members). At the end, instead of having two different Project classes or any other solution, the simplest one was having one attribute with the project's mode: work or personal.

- Asynchronous Data Retrieval: Dealing with asynchronous data retrieval from the Firebase database presented challenges, especially during page initialization or rebuilding when data changes occurred.

  Implementing a *refreshData* method and using listeners to the database helped in managing data updates effectively. Displaying a loading indicator during data retrieval helped improve the user experience.

- Continuous Improvement: Scrumer's development process highlighted the importance of iterative improvements and continuous refinement. Regularly evaluating the app's performance, identifying areas for enhancement, and incorporating new features or functionalities. For example the customization of the profile photo, was later added and it is a useful way for the user to identify his/her teammates in the project page or the task's comment section.

8.3 Future work

The idea of having a separate user subclass for each user type is based on making the software accessible for improvements in Scrum role-based features.

On the process of the goals establishment and requirement gathering, many ideas had to be abandoned either due to the complexity they posed or because they didn't align with the direction of Scrumer. One such idea, which could be a great new feature, is the addition of event tools where team members can join events. This could include timers,

displaying project members in meetings, or even collaborative activities like Scrum poker for sprint planning. Also, the feature of notifying the users when a sprint is finished or a task is done could also be added on the future. Furthermore, considering an advertising system as a potential business idea to generate revenue from Scrumer is also something to explore in the future.

In addition to introducing new features, the next step is to deploy the app on both the Play Store and the App Store.

8.4 Personal Conclusion

As a personal conclusion, the development of this project has resulted in a highly satisfactory outcome. Throughout the process, I not only learned how to build an app from scratch but also gained valuable experience in analysis, design, implementation and testing. One of the key lessons I learned was the importance of analysis and planning before starting the development process.

Additionally, I acquired new skills in Flutter and Dart, which are powerful tools for app development. Exploring these technologies introduced me to a vibrant community of developers. Learning a new language and framework from scratch expanded my technical knowledge and broadened my skill set.

Documenting the entire development process in detail was another valuable aspect of this project. It allowed me to practice and improve my documentation skills, which will be beneficial in future projects. Creating clear and concise documentation helped me summarize complex methods and processes, and present a comprehensive overview of the project.

Moreover, conducting this thesis as an exchange student in Germany, a different academic environment, enhanced my research skills and provided a more practical and less theoretical approach compared to what I might have experienced in my home university.

Overall, this project has been an enriching experience that has not only enhanced my technical skills but also improved my ability to plan, document, and execute complex tasks. It has provided valuable insights into app development, research methodologies, and project management.

Part IV

License and Bibliography

9 License Information

App License Information

Lottie Animations

The Lottie animations used in this app are public in the LottieFiles website. For them, the following license policy appears:

"Permission is hereby granted, free of charge, to any person obtaining a copy of the public animation files available for download at the LottieFiles site (“Files”) to download, reproduce, modify, publish, distribute, publicly display, and publicly digitally perform such Files, including for commercial purposes, provided that any display, publication, performance, or distribution of Files must contain (and be subject to) the same terms and conditions of this license."

These are the Lottie animations used:

- Normal user Registration option:

  https://lottiefiles.com/animations/man-working-on-laptop-in-office-IScjkzvNcq

- Work user Registration option:

  https://lottiefiles.com/animations/designer-team-at-work-Kd6ZuthqFV

- Create new project:

  https://lottiefiles.com/animations/startup-MZbGD6AQK5

- Create sprint:

  https://lottiefiles.com/animations/rocket-launch-E2fg4zXD59

- Sprint has no tasks (Colors have been modified):

  https://lottiefiles.com/animations/cronogram-peoples-6A2SRJaiWj

- Add Task:

  https://lottiefiles.com/animations/man-with-task-list-Z2XrIU43bQ

- Sprint not finished information (Animation has been modified):

  https://lottiefiles.com/animations/rocket-flighting-pqEmhIAhGC

- Sprint finished information:

  https://lottiefiles.com/animations/trophy-yEGPe40FVr

Flaticon assets

The photos used in this app are public in the Flaticon website. For them, the following license policy appears:

"Flaticon license: Free for personal and commercial use with attribution"

These are the photos and icons used:

- Scrum Master: https://www.flaticon.com/free-icon/delegation_9571752?term=coordinator&page=1&position=1&origin=style&related_id=9571752
- Product Owner: https://www.flaticon.com/free-icon/growth_7376385?term=developer&page=3&position=3&origin=style&related_id=7376385
- Developer: https://www.flaticon.com/free-icon/developer_2550601?term=developer&page=1&position=43&origin=style&related_id=2550601
- Normal User: https://www.flaticon.com/free-icon/leader_4059687?term=owner&page=1&position=13&origin=search&related_id=4059687
- Default User Profile image: https://www.flaticon.com/free-icon/user_219983?k=1686581730103&log-in=google
- Switch icon: https://www.flaticon.com/free-icon/transfer_10621360?term=exchange&page=5&position=17&origin=search&related_id=10621360

Packages

The packages (libraries, dependencies, etc.) used in this app are each subject to their individual licenses.These are the packages used:

- `cupertino_icons`: The MIT License text can be found at: https://pub.dev/packages/cupertino_icons/license

- `firebase_core`: The Apache License 2.0 text can be found at: https://pub.dev/packages/firebase_core/license
- `firebase_auth`: The MIT License text can be found at: https://pub.dev/packages/firebase_auth/license
- `google_sign_in`: The BSD 3-Clause License text can be found at: https://pub.dev/packages/google_sign_in/license
- `google_nav_bar`: The MIT License text can be found at: https://pub.dev/packages/google_nav_bar/license
- `cloud_firestore`: The BSD 3-Clause License text can be found at: https://pub.dev/packages/cloud_firestore/license
- `lottie`: The BSD 3-Clause License text can be found at: https://pub.dev/packages/lottie/license
- `uuid`: The MIT License text can be found at: https://pub.dev/packages/uuid/license
- `qr_flutter`: The MIT License text can be found at: https://pub.dev/packages/qr_flutter/license
- `qr_code_scanner`: The BSD 2-Clause License text can be found at: https://pub.dev/packages/qr_code_scanner/license
- `image_picker`: The MIT License text can be found at: https://pub.dev/packages/image_picker/license
- `image_cropper`: The Apache License 2.0 text can be found at: https://pub.dev/packages/image_cropper/license
- `firebase_storage`: The BSD 3-Clause License text can be found at: https://pub.dev/packages/firebase_storage/license
- `intl`: The Unicode License Agreement text can be found at: https://pub.dev/packages/intl/license
- `flutter_slidable`: The MIT License text can be found at: https://pub.dev/packages/flutter_slidable/license

- `flutter_countdown_timer`: The MIT License text can be found at: https://pub.dev/packages/flutter_countdown_timer/license
- `percent_indicator`: The MIT License text can be found at: https://pub.dev/packages/percent_indicator/license
- `flutter_localizations`: The BSD 3-Clause License text can be found at: https://pub.dev/packages/flutter_localizations/license
- `mockito`: The pache License 2.0 text can be found at: https://pub.dev/packages/mockito/license

Freepik image

One image from Freepik has been used for the default project image:

https://de.freepik.com/vektoren-kostenlos/flacher-desing-denkender-konzepthintergrund_4723691.htm#page=3&query=lightbulb%20project%20blue&position=29&from_view=search&track=ais

Document License Information

The images used for the profile pictures have been obtained from Freepik:

https://www.freepik.com

Library Image (other photos used):

https://static01.nyt.com/images/2015/10/24/opinion/24manguel/24manguel-superJumbo.jpg Back to School image (other photos used):

https://www.freevector.com

Other Apps Analysis:

Any analysis of other apps in this thesis (*Scrum App*, *Vivify* Scrum and *Toist*) is conducted solely for academic and research purposes. No unauthorized use or redistribution of the analyzed apps is intended or allowed.

Bibliography

[1] : *Adaptive and Responsive Layouts.* – URL https://docs.flutter.dev/ui/layout/adaptive-responsive

[2] : *Cloud Firestore.* – URL https://firebase.google.com/docs/firestore

[3] : *Cloud Storage for Firebase.* – URL https://firebase.google.com/docs/storage

[4] : *Dart Language.* – URL https://dart.dev/language

[5] : *Firebase.* – URL https://firebase.google.com/

[6] : *Firebase Authentication.* – URL https://firebase.google.com/docs/auth

[7] : *Firebase x Flutter Playlist.* – URL https://youtube.com/playlist?list=PLlvRDpXh1Se4Ceivpg8KrGvzb8BL9BHwo

[8] : *Flutter.* – URL https://flutter.dev/

[9] : *Flutter BottomSheet Widget).* – URL https://www.youtube.com/watch?v=pcHViPPbSHQ

[10] : *Flutter FAQ.* – URL https://docs.flutter.dev/resources/faq

[11] : *flutter_countdown_timer Documentation.* – URL https://pub.dev/documentation/flutter_countdown_timer/latest/

[12] : *How to Add App icons in Flutter | Automatic Manual Way 2021.* – URL https://www.youtube.com/watch?v=O9ChjwrZqns

[13] : *Large Scale Scrum: Comprehensive Overview of LeSS.* – URL https://www.digite.com/agile/large-scale-scrum-less/#coordination

[14] : *Listen to Realtime Changes with Cloud Firestore.* – URL https://firebase.google.com/docs/firestore/query-data/listen

[15] : *Miro*. Mobile Application. – URL https://miro.com/

[16] : *PERCENT INDICATOR Flutter Package of the Day*. – URL https://www.youtube.com/watch?v=nmkDW_cYrp4

[17] : *qr_code_scanner Example*. – URL https://pub.dev/packages/qr_code_scanner/example

[18] : *What is Scrum?*. – URL https://www.scrum.org/resources/what-scrum-module

[19] : *Todoist*. Mobile Application. November 18 2012. – URL https://todoist.com/

[20] : *Vivify Scrum*. Mobile Application. April 19 2019. – URL https://www.vivifyscrum.com/

[21] Flutter: *RefreshIndicator (Flutter Widget of the Week)*. – URL https://www.youtube.com/watch?v=ORApMlzwMdM

[22] Gupta, Pawneshwer: *How to open image with image picker, crop and save in Flutter*. – URL https://learnpainless.com/open-image-image-picker-crop-save-flutter/

[23] Heap, Richard: *Flutter: showDatePicker set first day of week to Monday*. – URL https://stackoverflow.com/questions/57975312/flutter-showdatepicker-set-first-day-of-week-to-monday

[24] Josephine, Maureen: *Generating QR Code in a Flutter App*. – URL https://medium.com/podiihq/generating-qr-code-in-a-flutter-app-50de15e39830

[25] Khan, Shaiq: *Slidable in Flutter*. – URL https://medium.flutterdevs.com/slidable-in-flutter-33193e2f1108

[26] Koko, Mitch: *Firebase Authentication in Flutter Playlist*. – URL https://www.youtube.com/playlist?list=PLlvRDpXh1Se4Ceivpg8KrGvzb8BL9BHwo

[27] Koko, Mitch: *Firebase x Flutter Playlist*. – URL https://youtube.com/playlist?list=PLlvRDpXh1Se4wZWOWs8yapI8AS_fwDHzf

[28] Koko, Mitch: *Flutter BottomSheet Widget*. – URL https://www.youtube.com/watch?v=pcHViPPbSHQ

[29] Koko, Mitch: *Modern Bottom Nav Bar*. – URL https://www.youtube.com/watch?v=FEvYl8Mzsxw

[30] Kukhnavets, P.: *What is Agile at Scale?* Hygger.io Guides. – URL https://hygger.io/guides/agile/agile-at-scale/

[31] Molloy, J.: *A Comprehensive Overview of the Client-Server Model*. – URL https://www.liquidweb.com/blog/client-server-architecture/

[32] Palacio, Marta: *SCRUM Master*. Iubaris Info 4 Media SL, June 2020. – Illustrations and cover: María de la Fuente Soro. Registered rights in Safe Creative. Registration number: 2006034305256. Iubaris Info 4 Media SL is the publisher and owner of distribution rights, released under the terms of the Creative Commons by-nd-nc 4.0 license.

[33] Srivastava, Naveen: *Date and Time Picker in Flutter*. – URL https://medium.flutterdevs.com/date-and-time-picker-in-flutter-72141e7531c

[34] Venema, M.: *6 Scaled Agile Frameworks – Which One Is Right For You?*. – URL https://www.nimblework.com/blog/scaled-agile-frameworks/

Personal Declaration of Authorship

Andrea Minguez Angulo, a student of the Bachelor of European Computer Science at Hamburg University of Applied Sciences, as the author of this academic document titled DESIGN AND IMPLEMENTATION OF A FLUTTER-BASED MOBILE APP FOR SCRUM PROJECT MANAGEMENT and presented as the Thesis for the Bachelor of European Computer Science.

I DECLARE THAT

This work is original and I have not copied or utilized any part of another work without clearly and accurately citing its source, both within the text and in the bibliography. I adhere to the principles of proper data usage and do not incorporate any third-party data without obtaining the necessary authorization, in accordance with the prevailing legislation. Moreover, I acknowledge that non-compliance with this obligation may lead to academic penalties, in addition to potential further consequences. In Hamburg, on 27th July 2023.

Signed: Andrea Minguez Angulo