Design and Implementation of Mobile Applications

POLITECNICO
MILANO 1863

BADGE-UP!

Design Document

Alessandro Mazza - 10670112
Adrian Valica - 10966734

Table of contents

Introduction

1.1 Description

The application, named **BADGE-UP!**, serves as a platform for knowledge sharing, progress tracking and skill planning. Users can create learning plans, share them with others, and embark on organized learning journeys to acquire new skills. The users will have a gamified experience, while progressing through the different Sections of the well-divided Skills created by themselves or by skilled community members.
Diving more into details:

- Each Skill, created by users, is composed of at least one Section, and each Section is composed of at least one Task.
- Users can create their own Skills or search for Skills that they are interested in and that are already published by other Users
- The User has a MySkills Screen where he can find all the Skills he decided to start. He can mark the accomplished tasks, and progress through the sections, while receiving badges as rewards.
- The user has the possibility of tracking his progress by checking the badges that he earned, and consulting his history, with all the Skill he finished, the Skills he created and the badges he earned, in a ordered manner.
- All these features are presented in a user-friendly manner, allowing fast and intuitive navigation though the application.

In this document we will now give an overlook on all the aspects of the application by specifying the features, providing an architectural description of the software, showing a general application workflow between the various screens and giving some notes on the implementation and testing processes.

1.2 Features

1.2.1 Sign in, Log in with different services and Update the Account

A User can sign in or create a new account by either using an email address or the Single-Sign-On functionality provided by identity providers such as

Google. He can then log into this account using the same credentials or Single-Sign-On to enter the application with his account. There is also a way for him to update his account, updating his username, password and profile picture.

1.2.2 Track general progress

A User can keep track of their progress with a statistics screen that shows them various information such as the obtained Badges, the percentage of completed Skills, the total number of Skills registered, started, or finished. It can get more details if needed, such as a complete dated history of his actions (skills created, finished, and badge obtained), and can consult the the descriptions of these events at any time in his progress.

1.2.3 Create a Skill

A User can create their own custom Skill using the Badge-Up application, by giving it a name, a description, at least one Section and flagging it either as *Private* or *Public*.
When adding a Section, the User needs to give it a name, a description. It also has to specify at least one Task that needs to be done a certain number of time to complete the Section. The user is free to add as many Tasks as he desires per Section, specifying for each a description and a number of times to realize.
During the creation process each Section and Task can be easily edited or removed.
The user gets notified before leaving the creation screen without saving, to be sure he doesn't accidentally loses his progress.

1.2.4 Search and Register new Skills

A User can easily search for new Skills to start, either from his Created Skills or from Skills created by other Users and flagged as *Public*.
When the desired Skill is found, the User can tap it to inspect all the related information and content of the Skill (name, description, Sections, and Tasks) and eventually register it to their collection of Skills he is interested in, to start them in the future.

1.2.5 Publish or Un-Publish a Skill

At any point in time, the user can select one of the skills he created and decide to Publish it, if it was a Private Skill, or to Un-Publish it if it was a Public Skill. This way, he can manage the visibility of his Skill to other user, allowing to test out a Skill before publishing it, or to Un-Publish an incorrect one.

1.2.6 Start a public or private Skill

At any point in time, the user can consult his Registered Skills and decide to start a Progression on any of them. He can then visualize it in his On-Going Skills and start realizing the Tasks.

1.2.7 Progress in a Skill

A User can easily browse between all the Skill they started in order to progress on the current Tasks and interact with them in multiple ways.

- When browsing the Skills, only the current Section is shown
- To progress a Task, just tap the respective button. Once the required amount is reached, the Task is automatically flagged as completed, and once all related Tasks are completed the Skill will automatically progress to the successive Section
- At any moment the User can diminish a Task progression, reset their progress on a certain Section, or on the whole Skill. Additionally, the User can also remove any Skill from their registered Skills if they wish to do so.

1.2.8 Create, Earn or visualize a Badge

When creating a Section, the User can decide to add a Badge to that Section, in order to make it more rewarding, add a symbolic value to the completion, and allow the user to store a Badge of this success in his badge page. He can choose between Bronze, Silver and Golden badges, and can add one per Section.
The user gets the badge when he completes the given section. A pop-up notifies him when he obtains this badge for the first time, and the badge is stored in his profile screen, where he can visualize them and the description anytime.

1.2.9 Order the skills in a custom way

As a user experience element, the user should be able to order the skills appearing on his started skills page in a saved way. This would allow him to have his skills organized as he prefers and make the search and progression easier for him when trying to mark a task as done.

1.3 Definitions and Acronyms

Here we list some useful definitions and acronyms that are used in the document as a reference

1.3.1 Definitions

General definitions

- **Android:** an operating system developed by Google meant for mobile devices and embedded systems in general, mainly smartphones and tablets. It is based on a modified version of the Linux kernel and other open source software
- **Framework:** an abstraction in which software, providing generic functionality, can be selectively changed by additional user-written code, thus providing application-specific software
- **Kotlin:** an open-source, statically-typed programming language that supports both object-oriented and functional programming that is officially supported by Google for Android development
- **Jetpack Compose:** an open-source Kotlin-based declarative UI framework for Android developed by Google, often used as a toolkit to simplify and accelerate UI development for Android.
- **Tier:** In computer programming, it refers to functional division of the software that runs on infrastructure separate from the other divisions. This way, we can have a software whose functions are distributed on multiple computers in a network.
- **Firebase:** a set of backend cloud computing services and application development platforms provided by Google. It hosts databases, services, authentication, and integration for a variety of applications, including Android

Skill specific definitions

- **Skill:** a series of multiple Tasks, divided in gradually more difficult Sections, all related to a similar topic. The can be *Private* (only the user who created them can access them) or *Public* (anyone can access them)
- **Section:** a part of a Skill that presents a set of Tasks that need to be completed. Once completed, it rewards a Badge to the user
- **Task:** a simple action that the user needs to do. Whenever the User does that action, they can simply tap the Task to increase their progress
- **Badge:** a reward given to the user for completing a Section, used to keep track of the total user progress

1.3.2 Acronyms

- **API:** Application Programming Interface, it indicates on demand procedure which supply a specific task.
- **DBMS:** DataBase Management System.
- **IdP:** Identity Provider.
- **MVC:** Model-View-Controller.
- **MVVM:** Model-View-Viewmodel.
- **SDK:** Software Development Kit: it provides a set of tools, libraries, relevant documentation, code samples, processes, and or guides that allows developers to create software applications on a specific platform.
- **UI:** User Interface.
- **S2B:** Software to Be, it is the one designed in this document and not yet implemented.

1.4 References

- Firebase documentation
- Jetpack Compose documentation
- Kotlin documentation

Architectural Design

2.1 Overview

This part of the document treats the Architectural Design of the Badge-Up application. It will dive deeper into the elements that compose the system and their interactions. We will analyse the choices that were made and explicit the reasons behind them in a detailed manner.

2.2 High-level View

The Badge-Up application is conceptualized with a clear separation of concerns. The main idea that we had in mind when conceiving this app is the creation of low-coupled and independent elements at all levels.
From a high-level perspective, this application is built using a Model View ViewModel (MVVM) Architectural Design. This separates the different components of the application into three categories, allowing for a better organization of the interactions between these high-level components. We will now describe each one of these elements and their function in the Badge Up Application.

2.2.1 View Elements

The View elements of the applications are all the Composable components and the interactions that they have with each other. We have two sets of screens serving different purposes:

- The Log-In and Sign-Up screens. These are the two first screens a user encounters, and their purpose is to authenticate the user. While being on this skill, the user only has access to the users-managing part of the application, and not to the skill-managing part by any means, thanks to the presence of two separate repositories. We will detail this separation when talking about the Model part of the architecture.
- The Profile, My Skills, Skill Search, and Skill Creation screens. These are the screens containing the core functionality of the application and are available to the user only once it is authenticated. In these screens only, the user has access to the features related to the skills, once he is authenticated.

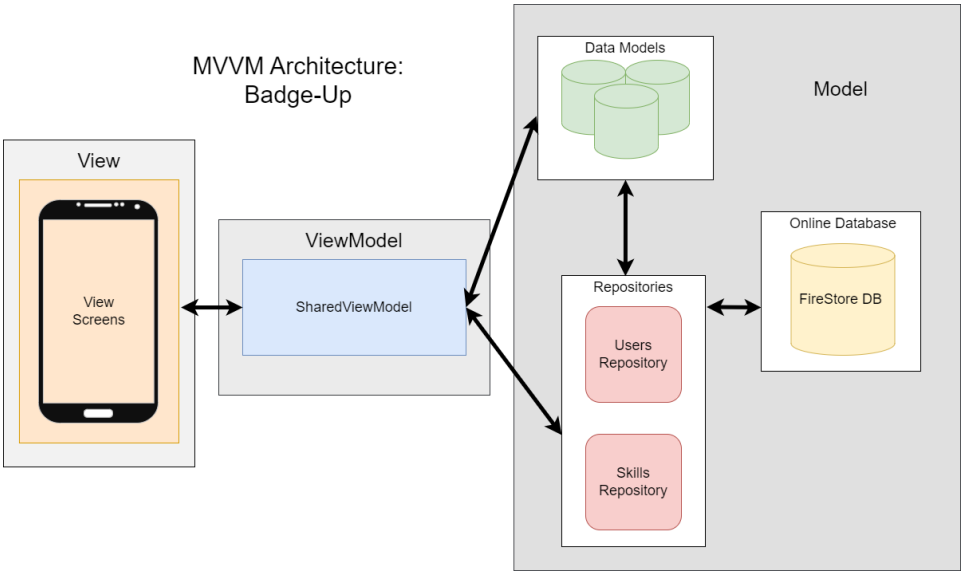


Figura 2.1: **High-Level View of the MVVM Architecture of Badge-Up**

2.2.2 ViewModel Elements

The ViewModel elements of this application are logic and runtime storage elements. After analyzing the requirements of our application, we decided that one single SharedViewModel would be the best choice for our application. It needs to store some data that should persist through the navigation, on different screens, and needs to present subsets logic functions from the same group.

The SharedViewModel stores information about the current user. This is useful to draw a connection between the Log-in/Sign-Up screens and the rest of the application, while at the same time keeping track of which user is logged in.

It also acts as a gap between the View and the Model part of the application, being the only entity that can access the repositories, and the database.

2.2.3 Model Elements

Since our application shall store the skills and present them to the user when needed, the Model part is the most complex one, and made of the largest amount of elements. We can draw three main components, being the Data Models, the Repositories, and the Database.

Data Model Elements Overview

The Data Models are some Parcelizable data objects that represent every aspect of the storable information used for the application. In a low-coupling-oriented approach, we decided to split them into three categories:

- The objects dedicated to one purpose. These are the objects that either store information only about the Users, or store information only about the Skills or Badges. The objects of the same category can be linked together, (for example, a Skill Section is linked to a Skill), but do not have fields referring to the other category objects, functionally. The models in this case are SkillModel, SkillSectionModel, SkillTaskModel, BadgeDataModel, and UserDataModel.

- The bridge objects, or relational objects. They are the objects that link the objects presented just before, and add some specific information to these relations. The objects in this situation are the SkillProgressionModel and the UserSkillSubsModel.

- The utility objects, are dedicated for runtime purposes. SkillCompleteStructureModel is not a data model that will be stored in databases, but is an efficient data structure, storing the precise information amount needed for the application to display the skills data, without storing too many useless objects.

With these three categories of objects, we could tackle three different areas of development. Changing the information stored in a SkillTaskModel for example, would not affect the UsersDataModel, and all the features using it would remain perfectly functional. At the same time, the relational objects only have the minimum amount of information to identify the two objects that they are linking, and therefore are not affected by changes in any of the data models.
The purpose of each DataModel will be detailed in the following section of the Document.

Repositories

The Repositories are links from the SharedViewModel to the Database system. They implement all the functions needed for data retrieval, saving, and updating. They store little data, making it available to the ViewModel. All the functions they implement are Coroutines, executed in the background, allowing the app to continue its normal functioning.
Again, for a separation of concern and easier development, the UserRepository and the SkillRepository have been separated, providing more modularity.

Online Database

The database we used to store all the data is the FireStore Database.
Since the goal of our application was to share the skill plans within the community, enabling communication on all learning subjects, an online database was necessary. We also considered creating a local database to store the information related to the user locally, however since the objects are always small we decided that there was no need for such saving. Most of the users will probably want to browse skills created by experts or professionals, therefore the online feature would be more present for their case.
The database stores all the previously stated objects except from the SkillCompleteStructureModel. It is accessed exclusively by the two repositories, with each interacting with distinct segments of the database collections to consistently maintain a singular version of the objects.

2.2.4 Interactions Between Elements

The interactions are simplified thanks to the MVVM Architectural Design. The View part (screens) can never access the Model part of the application directly, meaning that there is no interaction between Screens and Repositories or Online Databases. The opposite direction is also true.
When a given screen is required to access a data object, it interacts with the SharedViewModel, which is linked to the Repository and can retrieve, update, or save the given data object.

Model to ViewModel Interactions

Model-ViewModel interactions are the major part of the interactions. They can be classified into two types:

- Logical interactions. The View calls the logical functions from the SharedViewModel to compute results, conditions, or to modify data. An example would be the update of the skill progressions within the application. This occurs when a user interacts with the interface by tapping on a specific element on the screen, which in turn will increase their task advancement in the skill.

- The data management interactions. These are the interactions for which the View needs to retrieve data or launch a save/update of data depending on the actions of the user. An example would be when the user decides to save the skill he created to the database, in the Create Screen. This type of interaction uses both the SharedViewModel, but also the Model part since it needs the support of the repositories and database to be accomplished.

Some functions can combine both of the types, for example, a function computing which is the next SkillSection of a Skill can also retrieve its data.

ViewModel to Model Interactions

ViewModel-Model interactions are also very common. They are all the functions in the SharedViewModel that retrieve, save, or update data. They can only be called by the SharedViewModel itself. All these functions have the specificity to be executed in the background, while the application still runs normally. The three main operations are:

- The Save operation. This operation takes place when a new user registers to the application when he creates a new Skill, SkillSection, SkillTask, or Badge, or when he starts a skill. The Coroutine is simply launched by the repository and the application can continue its normal functioning.
- The Update operation. This operation is very similar to the save one, as it simply launches a save of an object that already exists in the database.
- The Retrieve operation. This operation is a bit more complicated than the two others. For this one, we usually want to execute an operation on the retrieved data. Therefore, the retrieval functions take as input which data to retrieve, but also a function to execute on the data once it is retrieved. This allows us to be able to let the application continue running while data.

2.3 Selected Database Management Choices

Now that we have seen how this application is using the MVVM Architectural design, we can dive deeper in the data models that have been explicit early on.

2.3.1 FireStore Implementation

FireStore is selected as our central database for all data models due to its real-time data synchronization capabilities and easy scalability. We have structured our database with a dedicated collection for each DataModel type, ensuring structured and efficient data retrieval and manipulation.

2.3.2 The DataModels

This subsection provides a detailed overview of each DataModel employed during the design of our application. These models were selected for their

efficiency, aligning with the requirements for swift and lightweight data transfers essential for an online-only database environment.

The UserDataModel

The UserDataModel is self-explanatory: it stores the data concerning the user. Each one of them has an email (which acts as the identifier, the primary key, with the constraint that one email can only be associated with one account), a username, and a Profile Picture, under the form of a URI. This UserDataModel is used in the Log-in, Register, and Profile screens, where the current stored user email is stored in the SharedViewModel to keep track of the connected user.

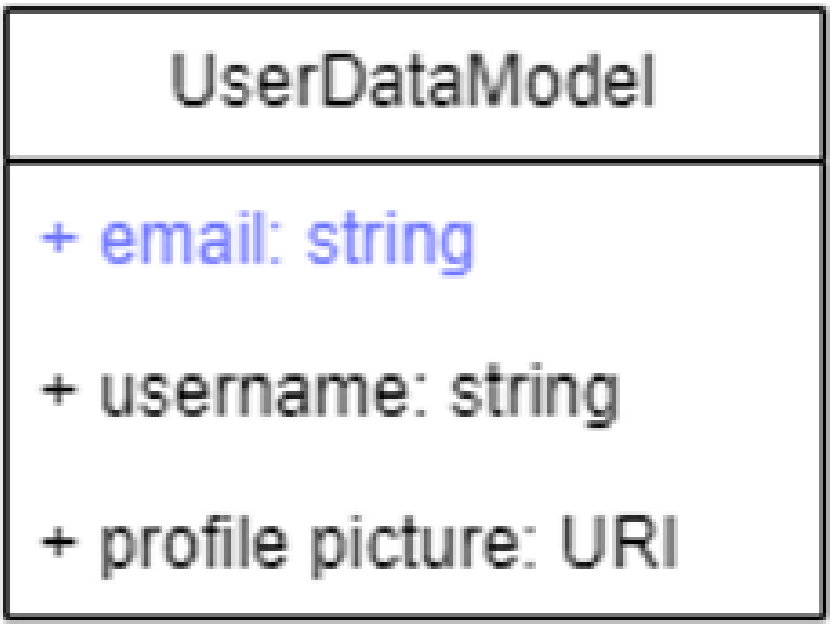


Figura 2.2: **Class diagram of the UserDataModel**

The SkillModel, SkillSectionModel, and SkillTaskModel

These three data models are linked together since they are the three pieces that constitute a skill.
On a high-level view, a SkillModel is made of SkillSectionModels, which themselves are made of SkillTaskModels. We chose to define these three data models in this precise way for two main reasons.

- The first reason is for Modularity and user freedom: we want the users to be able to create Skills of custom sizes, that would fit their needs. The user can add as many sections as he wants, and as many tasks as he needs for each of them, enjoying the flexibility of the application.
- The second reason is Hierarchy. Again, we want the user to be able to break a big problem into smaller steps, multiple times, which helps creating more comprehensive and easy-to-apply plans for learning Skills.

The SkillModel is the base of the skill, with its general information, and an ordered list of SkillSectionModels. We made this choice, in the spirit of the application, of achieving the goals step by step. Therefore, the SkillSectionModel is identified by a counter in the context of the SkillModel it belongs to. The SkillSectionModel itself has a list of tasks. Each of these tasks represents a goal that needs to be accomplished a custom number of times.

A task is said to be completed when it has been realized the required amount of times. A skill section is said to be completed when all of its tasks have been completed. When a skill section is completed, the application sets the skill progression of the user to start the next section. A skill is said to be completed when all the skill sections have been completed.

This division lets the user have access to two types of steps for his skills: an ordered one (sections) and a non-ordered one (tasks). It also allows him to have two different importance objects at his disposal to create custom Skills.

On a lower-level view this is the description of the three DataModels:

- The SkillModel: It is identified by a String ID (which acts as primary key). A SkillModel also stores the email of its creator, a title for the skill, a description, and a time when it has been created. It holds a list of the IDs of the sections that it is made of, in order to be able to retrieve them. Additionally, it stores a boolean telling if the skill is public, and the username of its creator, for more efficient display purposes.
- The SkillSectionModel: It is identified by the ID of its SkillModel, and an additional String ID identifying itself in the scope of the parent. It also has a title, a description, and a list of task IDs. In the scope of the Badge obtention, the section holds a boolean referring to the presence of a BadgeDataModel, and an ID to this potential BadgeDataModel.
- The SkillTaskModel: It is identified by the ID of its SkillModel, an ID of its SectionModel, and an additional String ID identifying itself in the scope of the parents. It also holds a description and an integer representing the required amount of times the task must be completed.

This is what these objects look like when put together in a diagram.

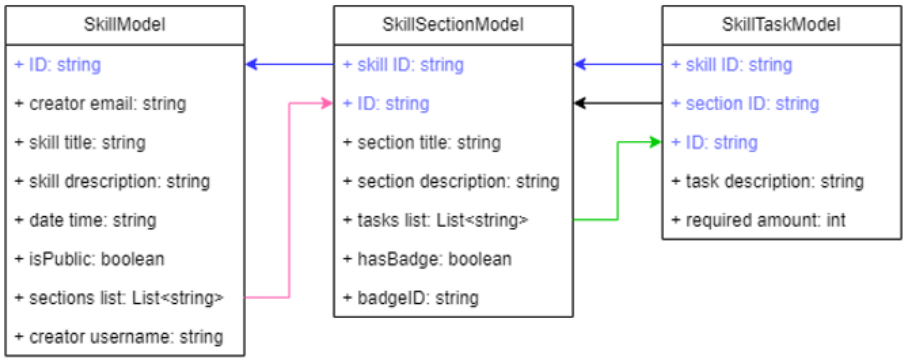


Figura 2.3: **Class diagram of the SkillModel, SkillSectionModel and SkillTaskModel**

The BadgeDataModel

This DataModel represents a Badge in the application. It acts as a symbolic reward for accomplishing an important section (such as a very challenging one, or a turning point in the progression of a Skill). The users can choose between three grades of badges: Bronze, Silver, or Gold. Therefore, they can be a symbol of success, allowing creators of skills to put more pressure on a section, and users of those skills to have a gamified experience, involving more motivation.
On a low-level view, a BadgeDataModel is identified by a Skill and Section IDs. It also stores the grade of the badge, its name, and its description.

The SkillProgressionModel

This SkillProgressionModel is a DataModel that represent the progression of a user in a skill. It is a transversal data model, linking a specific user to a specific skill and indicating what section he reached, and from this section which tasks were completed up to the current point in time. When a user progresses in the tasks, this DataModel is updated accordingly, allowing us to save the user progression.

The SkillProgressionModel is identified by a pair primary key made of the ID of the SkillModel concerned, and the email of the UserDataModel concerned. As information for the actual progression, it stores the ID of the current section, and a map linking the IDs of the non-finished SkillTaskModels to the number of times they have been realised up to that point. When a SkillTaskModel is completed, it is deleted from the list, and when the list is empty, the SkillProgrssionModel goes to the next section. There is also a

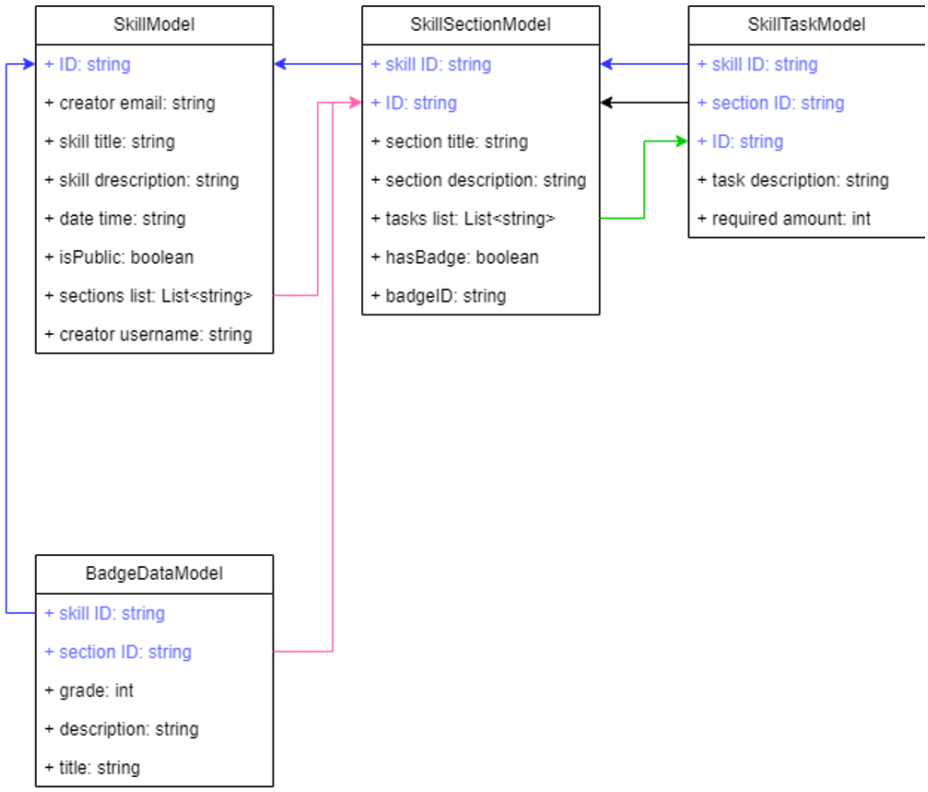


Figura 2.4: **Class diagram of the BadgeDataModel**

date and time for the moment when the SkillModel has been started by the user.

The UserSkillSubsModel

The UserSkillSubsModel is a DataModel completing the UserDataModel. It stores additional information that might be useful in some screens but is not always relevant when the UserDataModel is retrieved (such as the Register or Log-In Screens).
This DataModel stores personalized information about the user such as:

- The list of the IDs of the currently started SkillModels, and the time and date when they have been started.
- The list of the IDs of the SkillModels that the user has registered for later, and the time and date when they have been registered.
- The list of the IDs of the SkillModels that the user has finished, and the time and date when they have been finished for the first time.

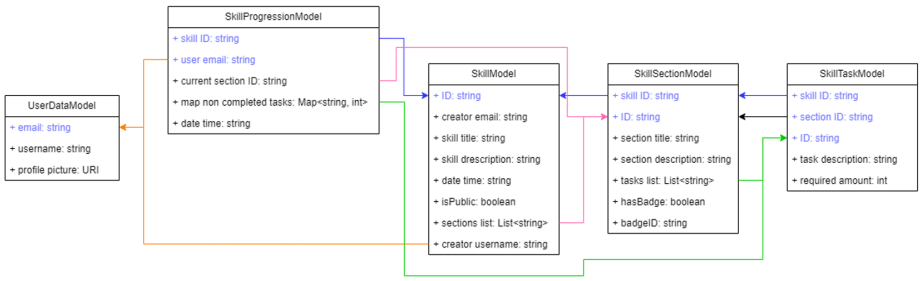


Figura 2.5: **Class diagram of the SkillProgressionModel**

- The list of the IDs of the SkillModels that the user has created.
- The list of the IDs of the Badges that the user has obtained, and the time and date when they have been obtained for the first time.
- The ordered list of the IDs of the SkillModels in the custom order that the user has selected, is to be displayed on the MySkills Screen.

This DataModel is identified by the email of the user, as the primary key.

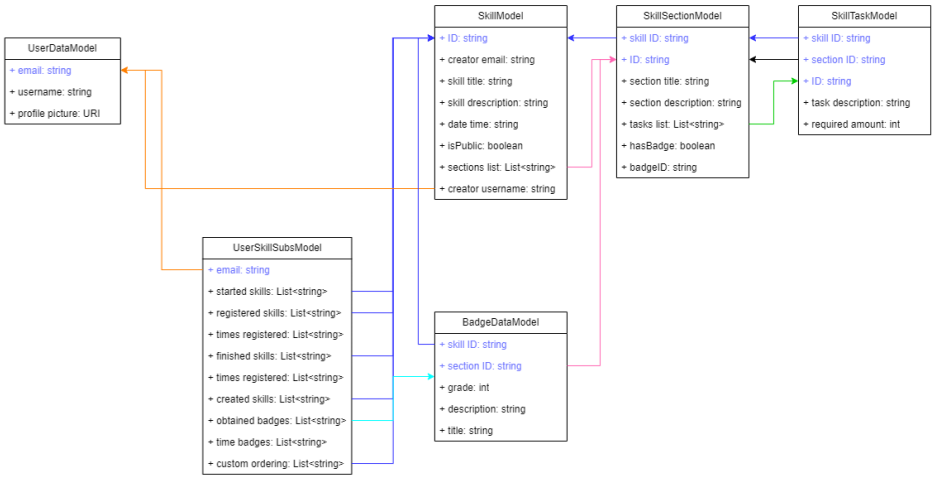


Figura 2.6: **Class diagram of the UserSkillSubsModel**

Putting Everything Together

Here is the complete class diagram of the DataModels once all put together, for having a better view.

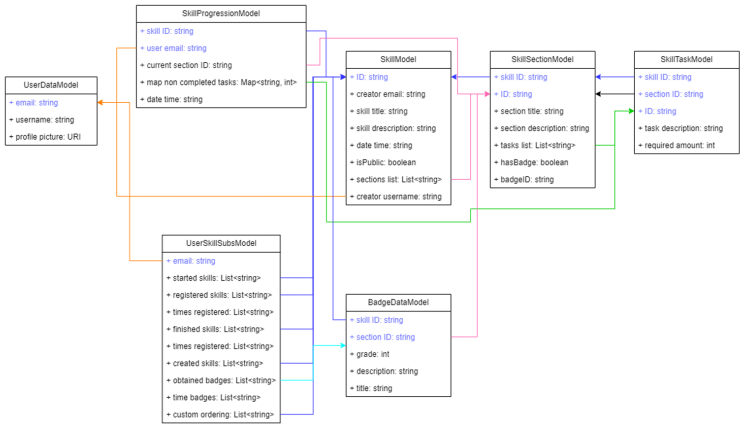


Figura 2.7: **Complete Class Diagram of the Database System**

This complex division of the elements allows us to have a simpler implementation of the different data models, which can easily be updated, and to give more freedom to the user with the shape of the Skills that he wants to create.

2.4 Selected Architectural Patterns

In this section, we will explicit the architectural patterns that have been used throughout the conception and the implementation of this application.

2.4.1 The Model View ViewModel Pattern MVVM

As said in the previous sections the MVVM pattern has been widely used in this application. The advantages that the MVVM pattern brings to our project is a separation of concerns and good testability, allowing us to test separately the View of the application, the logic through the ViewModel, and the database management through the Model part of the application. It also allows adding and updating features more easily.

2.4.2 State Hoisting

The application also implements the State Hoisting pattern, which separates the view elements responsible for displaying the application state from those computing the state. This separation is achieved by rendering interaction components as stateless and passing the resulting interaction functions as arguments to the component. Consequently, all these functions are defined within the calling component. This approach ensures that state management ascends the component hierarchy while state presentation descends.

2.5 Other Architectural Decision

This section will be presented some interesting decisions that were taken during the conceptualization and implementation of the application from an architectural point of view, and that are important enough to be presented.

2.5.1 Replication of the Data in Screens

Since all of our data management operations are done as Coroutines in the background, and retrievals of data take both time and bandwidth, we decided to implement a more efficient solution. Therefore, we replicated locally, into efficient data models, the data that we retrieve from the online database.

This way, when we need to update a datamodel, we call the SharedViewModel to update it online, which is not an instantaneous activity, while at the same time we update the local version of our DataModel to have a reactive display in our application.

For this purpose, we have created the SkillCompleteStructureModel which stores just enough memory to display all the skill content and the user's progression without having to retrieve data at each update of the SkillProgressionModel.

It stores a SkillModel, a SkillSection representing the current section that the user is doing, and a map for the skill tasks made of a SkillTaskModel and a Pair of two Integers (one for the current amount of times it has been realised and one for the required number of times). It also stores a SkillProgression that links for including the user concerned and the progression in the skill.

For example, every time a user marks a SkillTaskModel as "done once", the counter is incremented. From an online point of view, the function to update the SkillProgressionModel on FireStore is launched. From a local point of view, the SkillCompleteStructureModel is updated, with the new value of the SkillProgressionModel, and the SkillTaskModel list is also updated accordingly.

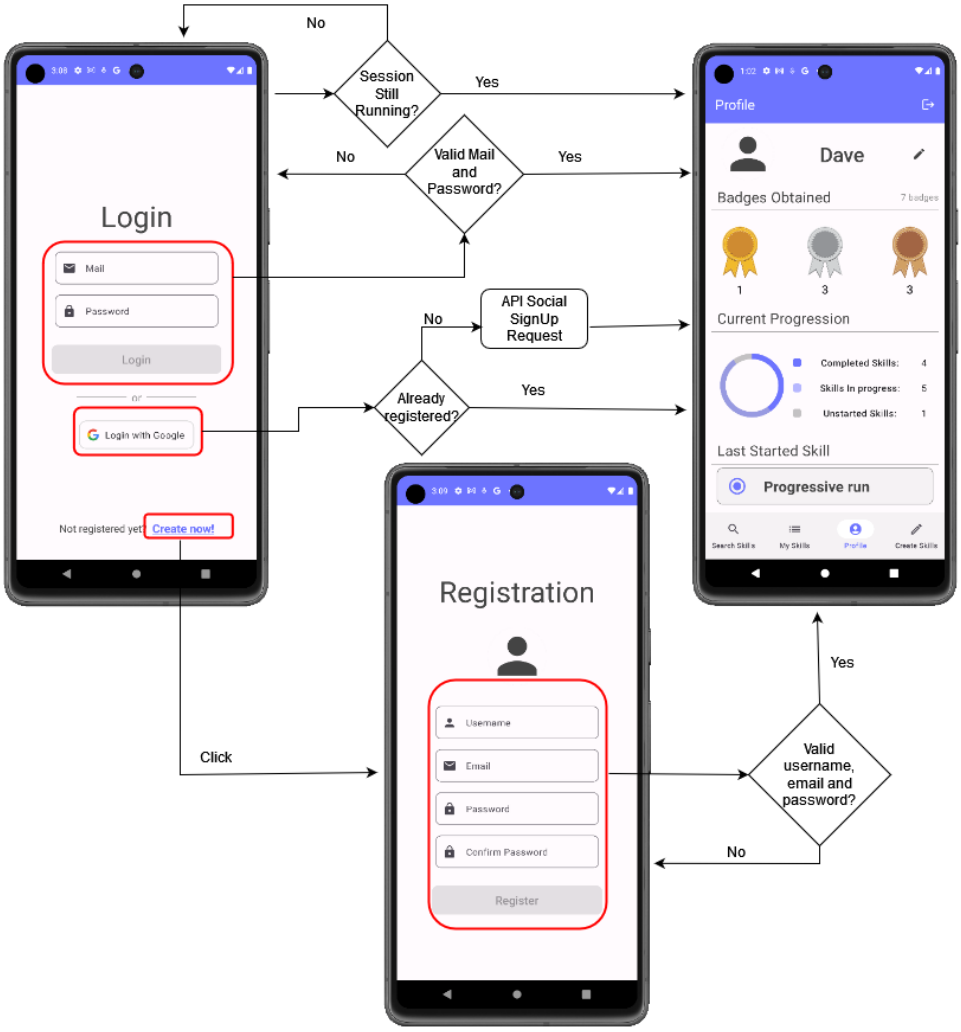
This way, when rebuilding the UI, the task and their progression can be loaded from the SkillCompleteStructureModel, even if the update of the online values has not been completed yet. This efficient implementation avoids unnecessary data retrieval, time delays on the application while ensuring data coherence between local and online storage.

User Interface Design

3.1 Screens workflow

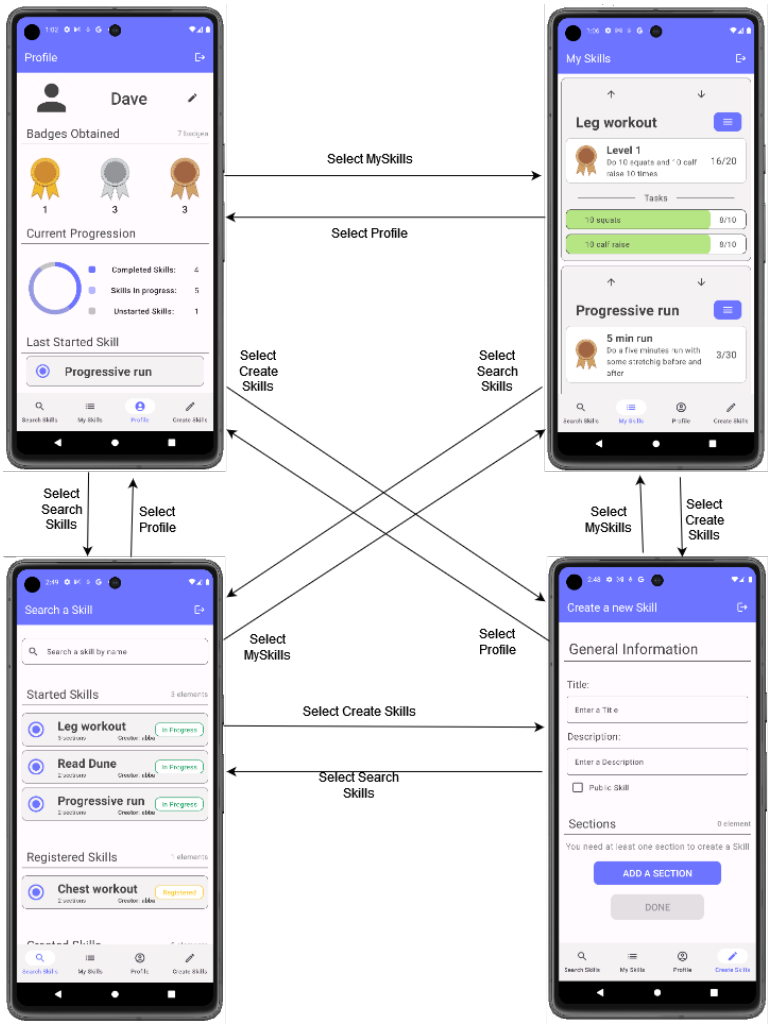
In this section we aim to give an overview of a general workflow of the main screens of the user application by representing how the main functionalities should be implemented and how they should work in the final product.

3.1.1 Sign in



When signing in, if the user didn't manually log out the last time they used the app they will reprise the current active session, authenticating automatically.
Otherwise, they can either use their registered email and password, or they can sign in with their Google Account.
If they wish to create a new account, the user can do so thanks to the Registration Screen.
Once they are logged in or they created a new account, the user will be automatically brought to their profile screen.

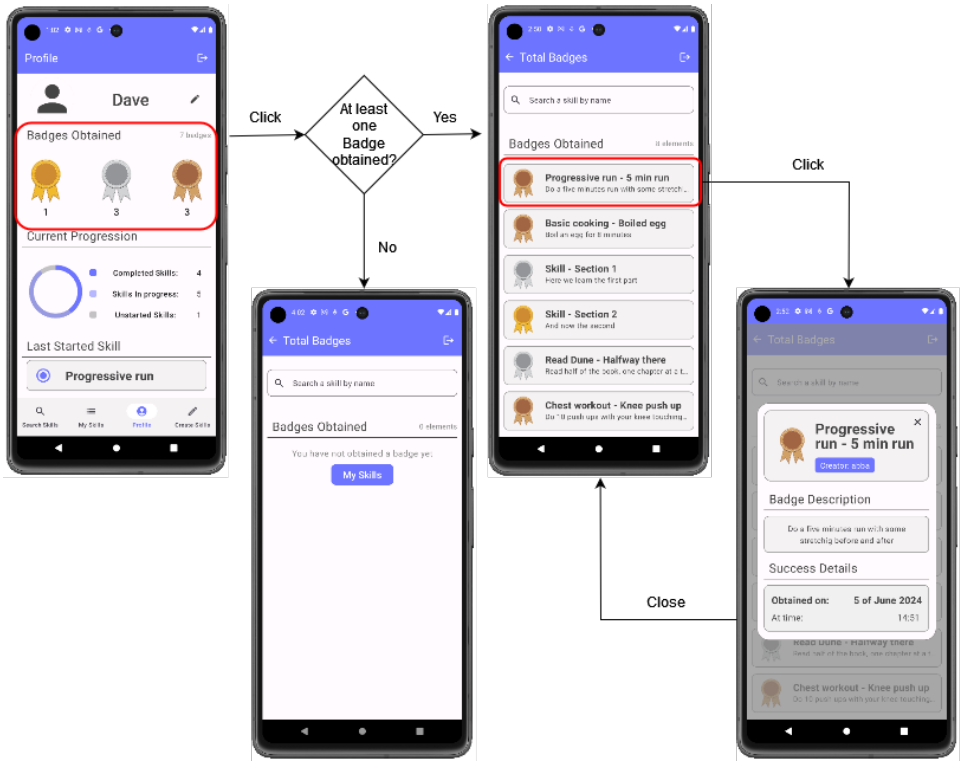
3.1.2 Bottom Navigation Bar



The user can easily cycle between the four main screens (Profile, Search Skill, Create Skill and My Skills) by tapping the respective button on the

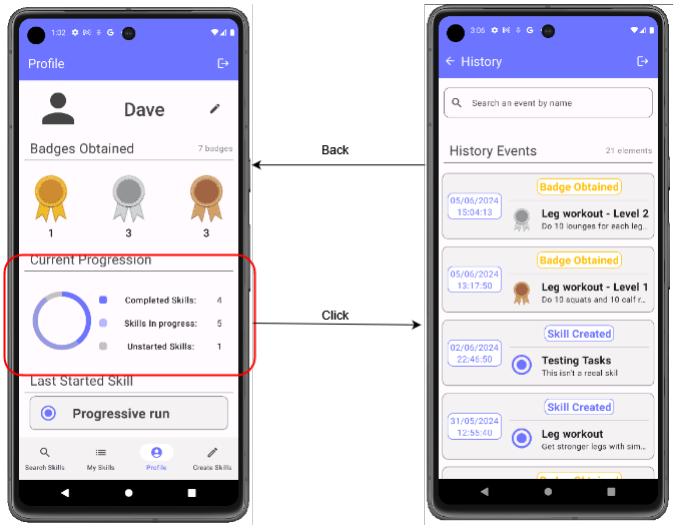
Bottom Navigation Bar.

3.1.3 Check Earned Badges



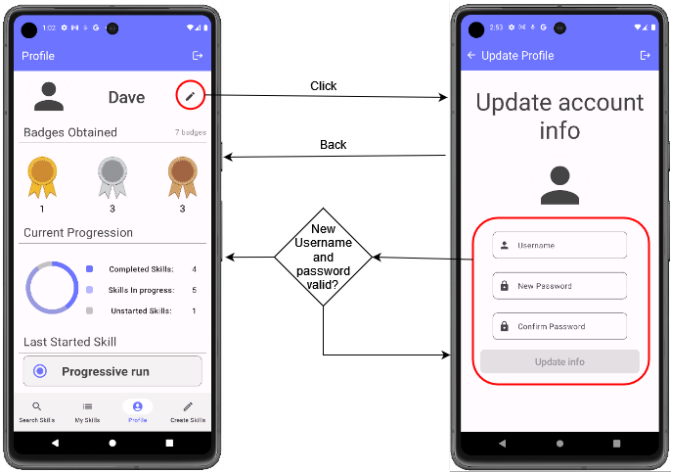
On the Profile screen, by tapping on the "Badges earned" section, the user can access the Badges screen, where they can look at all the badges they earned. Clicking on any of the Badge Banners will open a pop up that displays more info on the badge itself. Additionally, the user can search from all the earned badges with the search bar on top of the screen.
If the user has not obtained a Badge yet, the screen will instead show a text saying "You have not obtained a Badge yet" and a button that takes the user to the My Skills screen.

3.1.4 Check Complete History



On the Profile screen, by tapping on the "Current Progress" section, the user can access the History screen, where they can look at any milestone achieved (badges obtained, skills completed, skills created etc.) with a timestamp of the day and time they were achieved. They are ordered chronologically, and the user can search from all of them with the search bar on top of the screen.

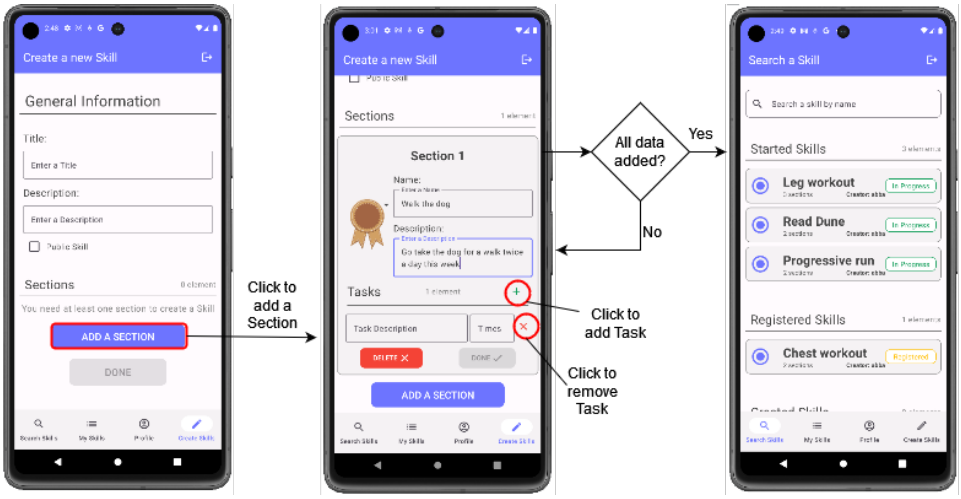
3.1.5 Update Profile



The user can update their Username, Profile Picture and Password at any time. They can access the Update Profile screen by tapping the Edit icon near their username in the Profile screen.

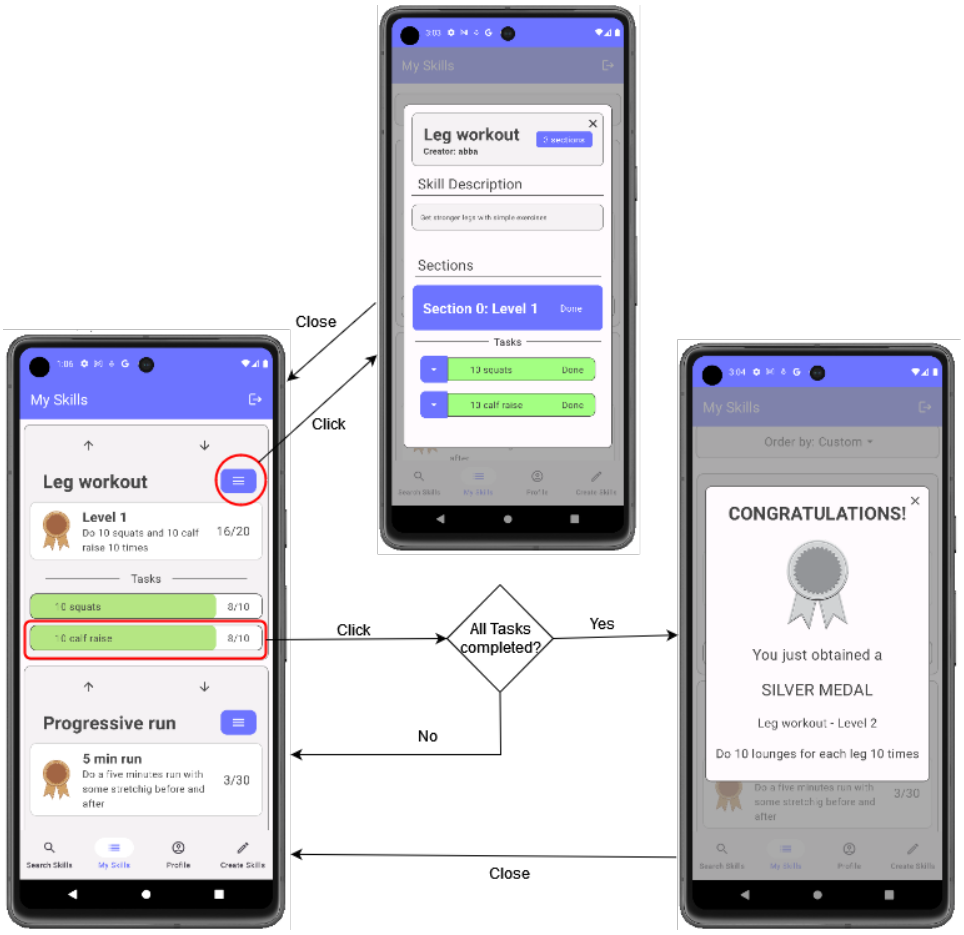
Once a new Profile Picture, Username and Password are selected and they are valid, they will be brought back to the Profile screen.

3.1.6 Create a Skill



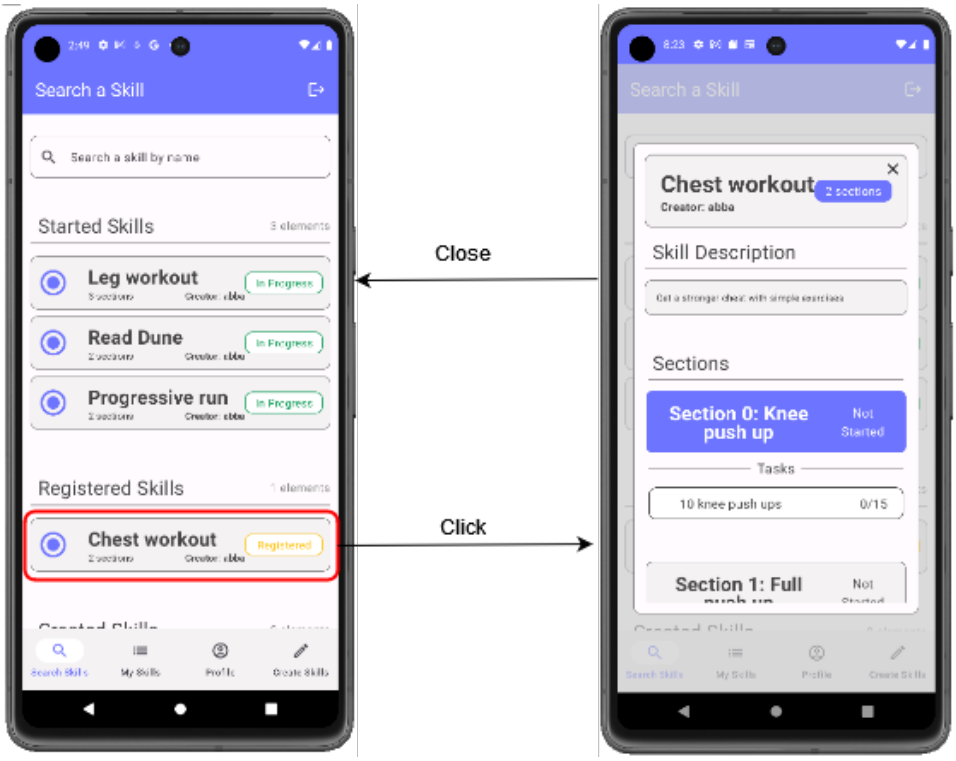
On the Create Skill screen, the user can create their custom skill by specifying the name, the description and any custom section the user wishes to add. By clicking the "Add a section" button, a new Section UI appears on the screen that allows the user to customize the section by choosing a badge type, a name, a description and any task they wish.
Once the Skill is completed, the user will be taken to the Search Screen.

3.1.7 Skill Progression



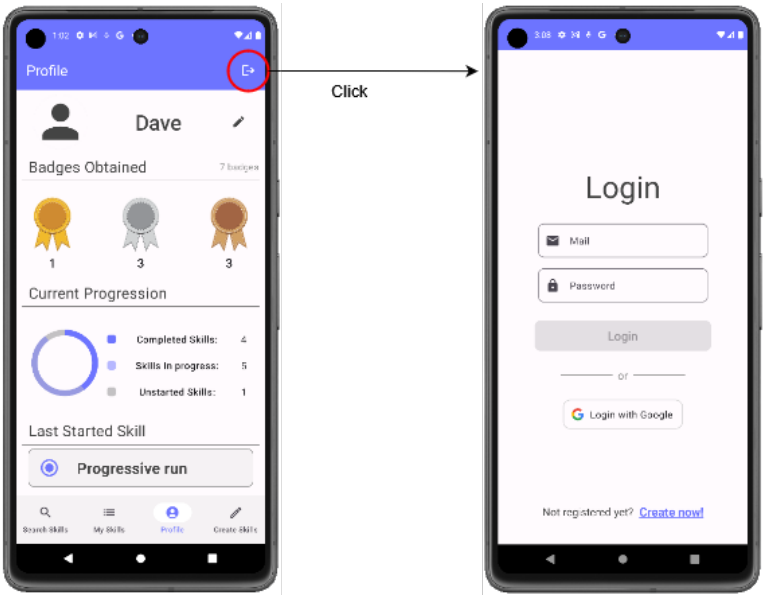
On the My Skills screen, the user will be presented to a list of UI elements that show the current Section of each Skill that is in progress.
On top of the screen, there's an option to reorder these elements either chronologically (ascending or descending) or with a custom order.
Each elements present the name of the current Section, all the related Tasks and the related badge. A button on the top right corner allows the User to open a scrollable pop up that shows more info on the whole Skill, allowing the User to see all Sections, restart any completed Section or even the whole Skill . If all tasks are completed, a pop up with the Badge that was just achieved is shown to the user.

3.1.8 Search a Skill



On the Search Skills screen, the user will be presented with all the skills they saved, divided in "Created skills" (gathering all the Skills the user created), "Registered Skills" (gathering all the Skills you saved but not started) and "Started Skills" (gathering all the Skills currently in progress). If the user wishes to look for new Skills, they can choose to browse between the Skills created by other users.
By clicking on any of the Skills shown, a pop up with all the information related to the Skill (name, number of section, creator, description, tasks etc.) shows up, giving the option to save a Skill or starting it (if not saved) Additionally, the user can directly search for a specific skill by name with the search bar on the top of the screen.

3.1.9 Log out

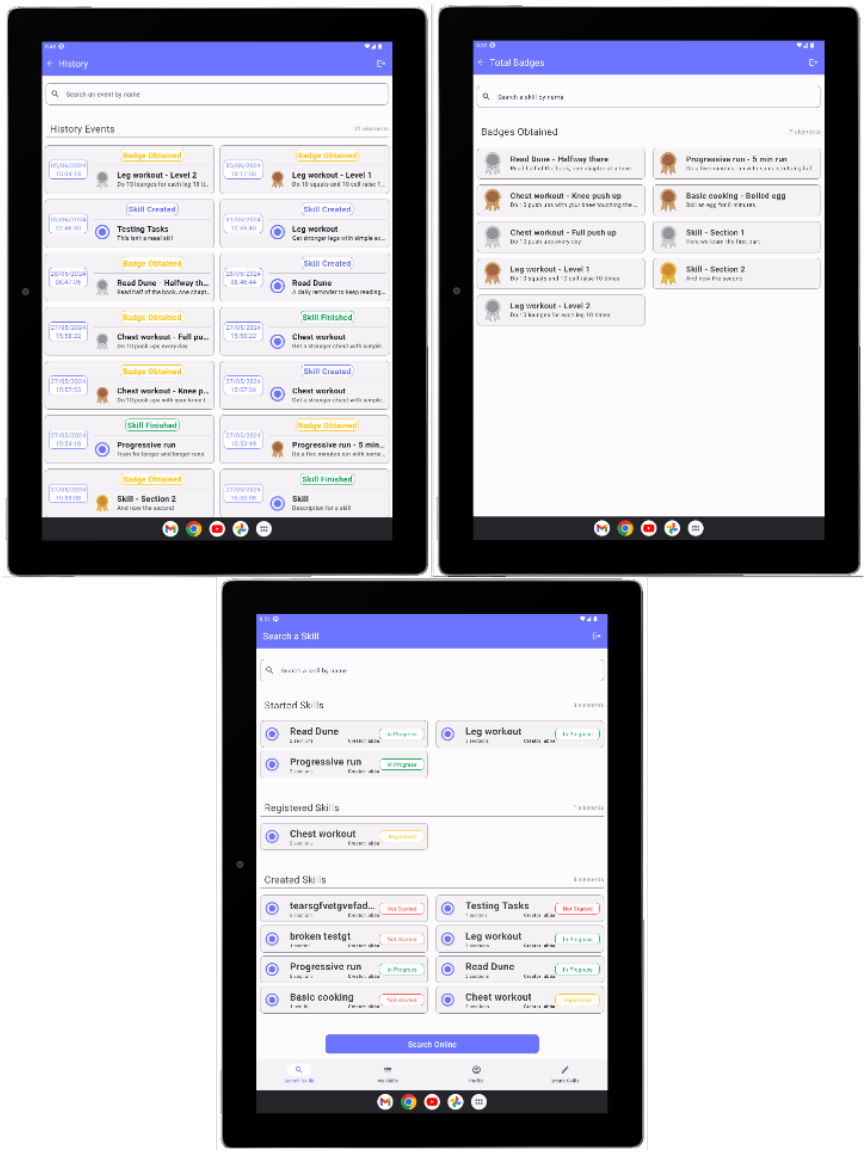


In most screens the app presents a top app bar with a Log Out icon. Tapping it will end the session, logging the user out of the app and taking them back to the Login screen.

3.2 Alternative UI

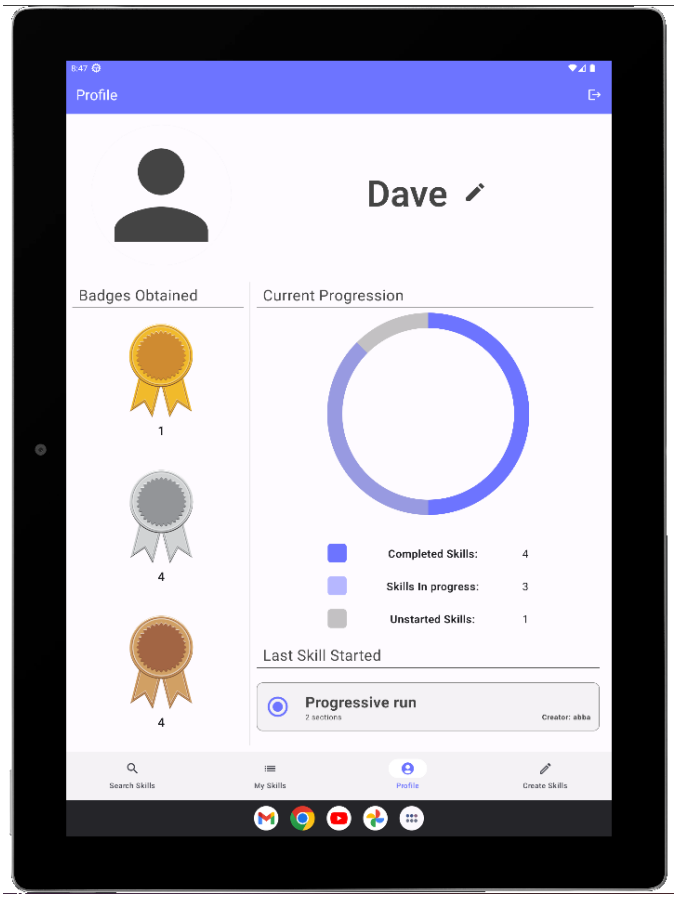
To better fit the larger screen of tablets, many UI elements are resized or have different proportion. For some Screens, an alternative UI that better utilizes the larger spaces is automatically used instead.

3.2.1 Search, Badges and History screen



To better fit the wider screen, all Screens that feature a list of elements displays them in two columns, making it easier to browse between them and keeping a proportioned look.

3.2.2 Profile screen



On the Profile screen, the Badges earned section is now displayed vertically and more space is given to the chart in the Current progression section

3.3 User Experience Principles

When building the Application Interface, we also made sure to respect some well known principles of design that would better the User Experience. We mainly applied some of the Nielsen Heuristics:

- User Control and Freedom: We made sure that the user can cancel any pop up or situation intuitively, by either providing a cross in the upper right corner or an explicit Close button. Going from one screen to the other is simple, with the bottom navigation bar, and the fact that this is the only navigation way, makes it easier for the user to navigate

through the app. Our application supports the "Back" action from the phones, allowing them to go back to a previous screen.

- Consistency: We made sure that the application stays coherent in its presentation. All the texts are of the same color and same font, and we use a small set of font sizes for the different purposes in our application (text, title, subtitle...). We use a color palette of 4 colors, which are always the same throughout the application. The layout of the titles of Screen sections is always organized in the same way. The visualizations are all made through pop-ups that can be navigated by the user in the same way.

- Flexibility and Efficiency: We let the user organize their In progress skills list in the order they prefer, and this order is saved, being restored every time they open the application. All of the screens allowing for the user to search have a search bar, to help the user target the desired element more efficiently.

- Minimalist and sober design: We made the choice to use a minimalist design, to help the user be more focus on his task. There are no invasive images, all the shapes are Rounded Corners to feel warmer and more appealing for the user to progress through the skills, without being disturbed. The font is Dark Gray instead of Black, to have lighter pressure on the user's eye, and the white background doesn't distract the user from his tasks.

All these details and choices aim to bettering the user experience and its feel when using the application.

Implementation and Testing

In this chapter we aim to give an overall look on the overall implementation of the S2B by considering both the current prototype and any future plan for a complete implementation and release to the market

4.1 Implementation and Testing Plan

4.1.1 The Approach

In this section, we will present the plan and the strategies we followed throughout this project's development and testing phases. Since our application is tackling multiple separate functionalities and features, as presented in the first section of this document. Therefore we adopted a combination of two implementation and testing approaches:

- A bottom-up approach. Since multiple base elements are used for multiple features, a bottom-up approach suits well the Badge-Up project, by first implementing and testing the core components, on which will rely bigger ones. This structural approach would ensure modularity, and provide continuity in testing, since the components can be tested as soon as they are implemented, building a solid basis for the greater ones by enabling easy bug tracking.
- A thread-based approach: this approach is important for being able to showcase early results, to test user experience, and be able to pivot sooner if the direction taken seems to be wrong. We would therefore start implementing the functionalities one by one, to have a first version as soon as possible to test out with real users.

The combination of these two approaches results in a strategy consisting to implement and test each feature in a bottom-up manner, allowing for feedback, modularity, high-testability and a high pivoting speed.

4.1.2 Feature Identification

The first step was to identify the main features of our application. Being already stated in the first section of this document, we will list them again, then explicit the actual implementation plan.

[F1]: Sign in, Log in with different services, and Update the Account: This feature is the core of the Account management part of the application. It mainly involves the Sign-Up, Log-In, and Profile screens. From a Model point of view, it will use the UserDataModel. The SharedViewModel must be adapted to being able to save, retrieve, and update elements from the Users collection of the Firestore Database.

[F2]: Create a Skill: This feature is a core functionality of our app, since it revolves around creating Skills for the user itself, or for other users to take profit from them. It will involve the Create screen from a View aspect. From a Model point of view, it will require the SkillModel, SkillSectionModel, and SkillTaskModel mainly.

[F3]: Search and Register new Skills: This feature comes important to be able to find all the skills that have been created, both by the user and by other users if they are published. It takes into account the possibility to browse the skills, to display their contents, and to be able to be registered. Therefore, the Search Screen is the one that will be affected. From a Model point of view, it will require the SkillModel, SkillSectionModel, and SkillTaskModel and the UserSkillSubsModel.

[F4]: Publish or Un-Publish a Skill: This feature is an add-on to the [F3]. It adds a way for a user to publish or unpublish the skills from the ones he created. This will happen in the Search screen and will require the SkillModel.

[F5]: Start a public or private Skill: This feature allows the user to start his progression on a Skill. It will be present in the Search Screen, and will mainly require the SkillProgressionModel from a model point of view.

[F6]: Progress in a Skill: This is a core feature of our application since it comes after the feature [F5]. The user should be able to mark his progression in realtime on the tasks of a skill and to be able to advance to the next sections. This will take place in the MySkills screen and will require the SkillProgressionModel.

[F7]: Create, Earn or Visualize a Badge: This feature is the gamification part of our application, since it will add an additional reward to the user. It will be present in the Create, MySkills, Profile and Badge Screens. From a Model point of view, it will require the BadgeDatModel, the UserSkillSubsModel, and the SkillSectionModel.

[F8]: Track general progress: This feature is one of our main focuses, to give a better insight to the user about what he has realized up to now, regrouping the results of most of the precedent features. It will involve the Profile, History and Badge Screens. From a Model point of view, it will be using all of the DataModels that have been used up to now and the UserSkillSubsModel.

[F9]: Order the skills in a custom way This feature is a user experience add-on. It will be present in the MySkills Screen and will only require the UserSkillSubsModel.

Having all these features, we decided to split them into Flows, which are user-testable functionalities, regrouping multiple Features and that can act as a Thread in our thread-based approach.

Flow 1: User Signs In, Logs In, Updates its account and Signs Out This first Flow is a good use case to test all the UserAccount parts of the application. It targets mainly the feature [F1].

Flow 2: The User Creates a Skill and Visualizes it in the Search Screen This second Flow will target the features [F2] and [F3]. It will allow us to have feedback on the creation screen, and the intuitiveness of the Search Screen, and to test the linking between the Create Screen and the Search Screen.

Flow 3: Publish a skill, Register a public skill, Start it, and Order it in a custom order This Flow will concern the situation when there are multiple Users. It will target the features [F3], [F4], [F5], and [F9] allowing us to implement and test the public aspect of the skills, and their visibility between the different user accounts.

Flow 4: Progress through a Skill, reset Progression, and Finish the Skill This Flow will only need one user. He will have to be in a situation where a skill is started (which has been verified to be working in the previous flows), and he will have to progress through the skill, by increasing the tasks and the sections as he goes. He will also be testing out the reset functionalities and the Finishing of a Skill. This Flow will target the features [F5] and [F6].

Flow 5: Create a Skill with a Badge, Start it, progress through it and Finish it This Flow will test out multiple of the features from the anterior flows: [F2], [F3], [F5], [F6] but will also include the features [F7] concerning the creation and obtaining of a Badge.

Flow 6: Visualize the progression, the History, and the Badges Obtained This Flow will come after the others so that the user has some Skill finished, and badges obtained. This will test the intuitiveness of the Profile, Badge, and History page. It will target the features [F7] and [F8].

We will be using these Flows for the integration and testing part of our documents since they constitute threads that can be shown to obtain feedback, and since implemented in a bottom-up fashion, granted us an efficient bug tracking phase.

4.1.3 Component-Level Integration and Testing

The component integration and testing part of the Badge-Up project ensures that every component works alone before integrating with the other components to implement one of the features. We used a cyclic methodology for this step, always starting from the Model part of the Flow, the adapting the SharedModelView and the FireStore Database if needed, and finally updating the Views. Here is the detailed plan for our implementation.

Flow 1: The User components

The Flow 1 integration is depicted in Figura 5.1. The first step was to create the Model part, more precisely the UserDataModel (green in the figure). Then we adapted the FireStore database (yellow in the figure), the SharedViewModel (orange) to be able to retrieve and save data concerning the Users. In the end, we created the view part, with the three screens that were needed: Sign Up, Log In and Update Account (blue).

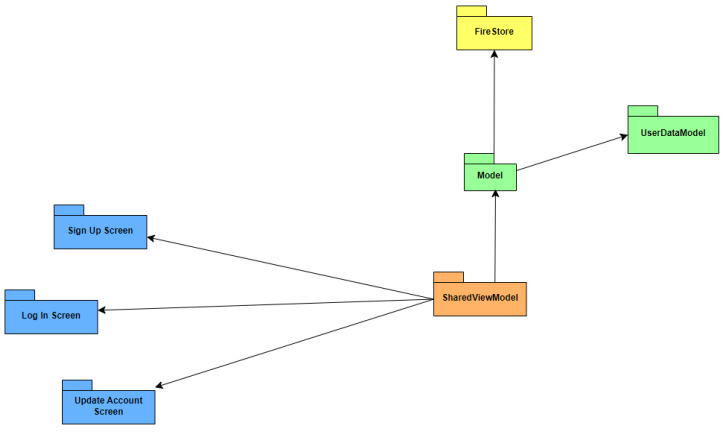


Figura 4.1: **Component-Level integration for Flow 1**

This implements the [F1] feature correctly, allowing us to showcase the possibility for the user to register, login, and modify his account.

Flow 2: The Skill Creation and Visualization components

The Flow 2 integration is depicted in Figura 5.2. The first step was to create the Model part, more precisely the SkillModel, SkillSectionModel and SkillTaskModel datamodels (green in the figure). Then we adapted the FireStore database (yellow in the figure), the SharedViewModel (orange) to be able to retrieve and save data concerning the Skills. In the end, we created the view part, with the parts of the two screens that were required: Create Screen and Search Screen (blue). For the Create Screen, the view was complete and functional, except for the Badge creation and the public flag. For the Search Screen, we had the listing of the created skills and the visualization pop ups.

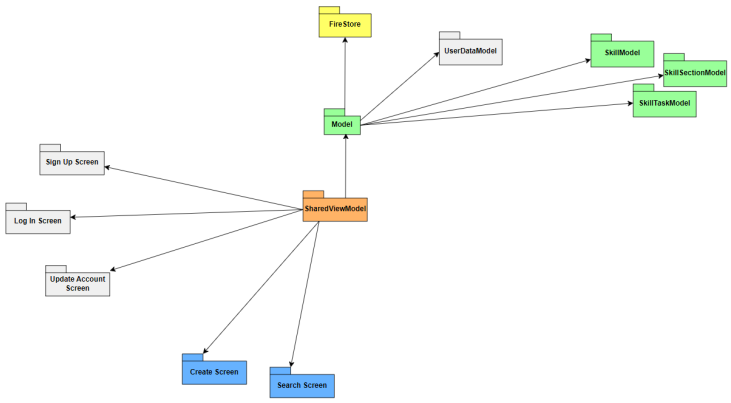


Figura 4.2: **Component-Level integration for Flow 2**

This allowed us to showcase the features [F2] and a part of [F3].

Flow 3: The Publishing and Starting Components of the Skills

The Flow 3 integration is depicted in Figura 5.3. The first step was to create the Model part, more precisely the UserSkillSubsModel data model only (green in the figure), since the SkillModel, SkillSection, and SkillTask were already implemented. Then we adapted the FireStore database (yellow in the figure), the SharedViewModel (orange) to be able to retrieve and save data concerning the Skill registration, starting, and the custom orderings. In the end, we created the view part, with the parts of the three screens that were required: the Search Screen, the Create Screen and the MySkill Screen(blue). For the Create Screen, the view was updated to allow the possibility to mark a skill as public or not. For the Search Screen, the view was updated to allow registration of skills, starting of skills, and to allow the user to browse the published skills. The possibility to publish or unpublish a skill was also added. For the MySkills Screen, we had to create

the screen and allow the display of these, as started. We also had to include the possibility of changing the order of skills.

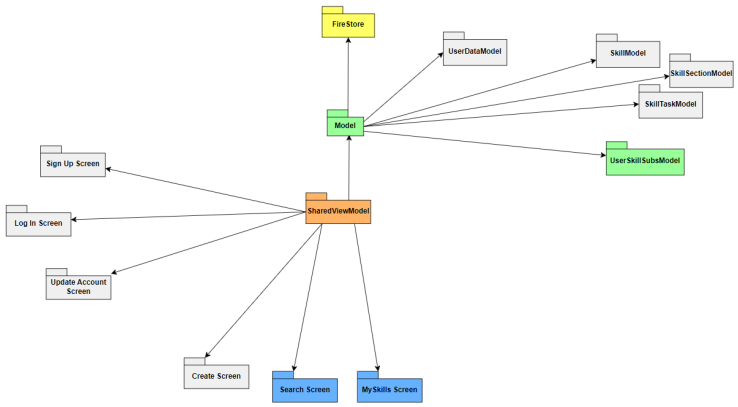


Figura 4.3: **Component-Level integration for Flow 3**

This allowed us to showcase the features [F3], [F4], [F5], and [F9].

Flow 4: The Progression Management Components of the Skills

The Flow 4 integration is depicted in Figura 5.4. The first step was to create the Model part, more precisely the SkillProgressionModel and update the UserSkillSubsModel data models (green in the figure). Then we adapted the FireStore database (yellow in the figure), the SharedViewModel (orange) to be able to retrieve and save data concerning the Skill progressions and completions. In the end, we implemented the view part, with the parts of the MySkill Screen that needed to be updated (blue). We implemented the parts of the screens that allowed the user to increment and decrement a task, the possibility to reset the progression

This allowed us to showcase the features [F5] and [F6].

Flow 5: The Badge Creation, Obtaining and Visualization Components

The Flow 5 integration is depicted in Figura 5.5. The first step was to create the Model part, more precisely the BadgeDataModel and update the UserSkillSubsModel and the SkillSectionModel data models (green in the figure). Then we adapted the FireStore database (yellow in the figure), the SharedViewModel (orange) to be able to retrieve and save data concerning the Badges creations, the corresponding SkillSections, their obtaining, and their visualization. In the end, we implemented the view part, with the parts of the MySkill and Create Screen that needed to be updated, plus the Profile and Badge Screens that had to be created (blue). For the MySkills

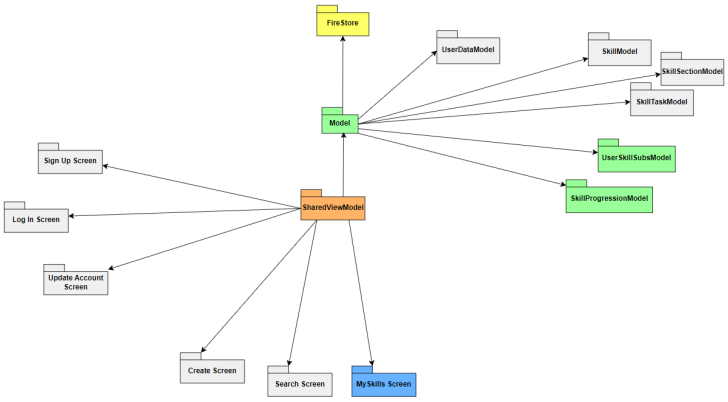


Figura 4.4: **Component-Level integration for Flow 4**

Screen we added the pop ups to notify the user when he obtains a badge. In the Create Screen, we added the possibility in each Section to add a badge. We created the Profile Screen that displayed some information about the user progression and a section that showed the amount of badges that have been obtained. We also created the Badge Screen that showed a list of all the badges that have been obtained, and could provide more information about them all.

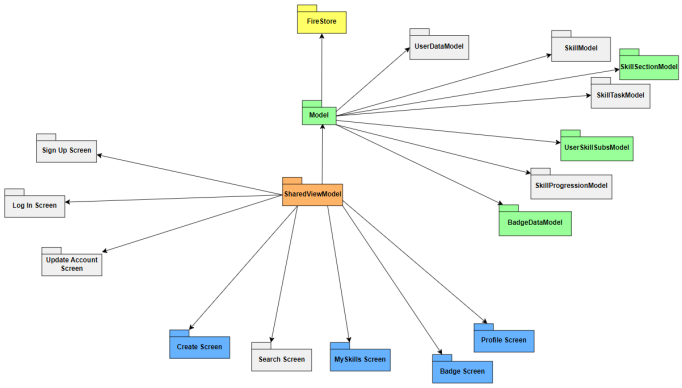


Figura 4.5: **Component-Level integration for Flow 5**

This allowed us to showcase the features [F2], [F3], [F5], [F6] and [F7].

Flow 6: The Evolution Tracking Components

Lastly, the Flow 6 integration is depicted in Figura 5.6. The first step was to create the Model part, more precisely to update the UserSkillSubsModel data model (green in the figure). Then we adapted the FireStore database (yellow in the figure), the SharedViewModel (orange) to be able to retrieve and save data concerning the times of the events such as finishing skills, obtaining badges, and creating skills plus thee visualizations of all these elements. In the end, we implemented the view part, with the parts of the Profile and Badge Screen that needed to be updated, plus the History Screens that had to be created (blue). For the Profile Screens, we added some statistics about the number of started, finished and registered skills that the user had. For the Badge Screen, we added the information about the time and date when the badges were obtained. We created the History Screen by taking all the stated events and displaying them in a chronological order, while letting the user get detailed information about any of them by clicking on them.

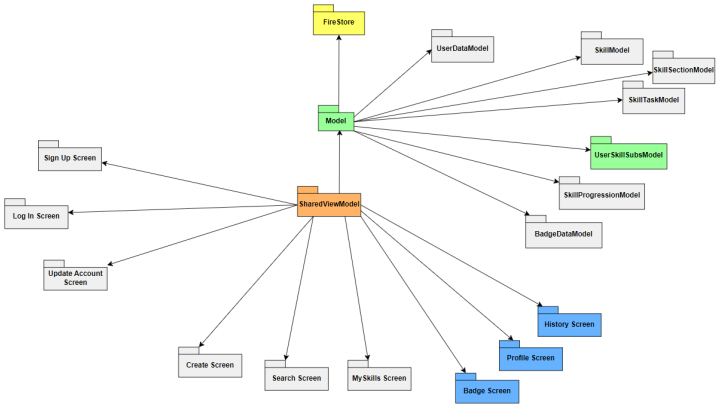


Figura 4.6: **Component-Level integration for Flow 6**

This allowed us to showcase the features [F7] and [F8].

The Complete Component View

Here is the diagram representing our complete Component View at the end of the Integration and Testing. The Model components have been marked in green, the View ones have been marked in Blue, the SharedViewModel in orange, and the FireStore external database in yellow.

This approach shows how we were able to make use of the separation of concerns and of the MVVM pattern to implement the application. The data models are always implemented first, and the views at the end, after adapting the SharedViewModel to provide them the required logic functions

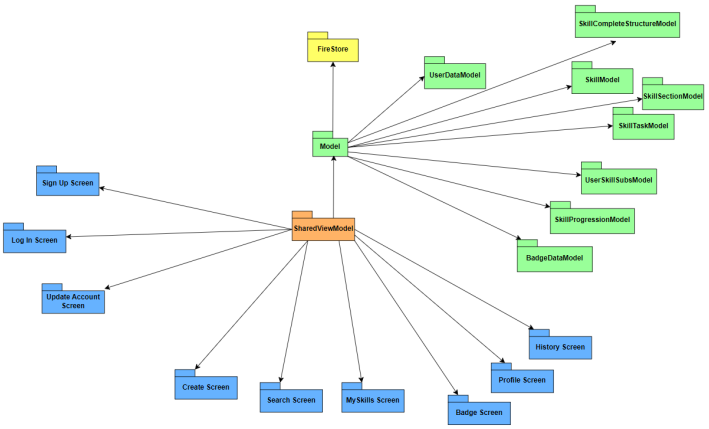


Figura 4.7: **Complete Component View of the Badge-Up Project**

and ways to allow them to retrieve and save data. If one of the data models were to change, only the SharedViewModel would need to be changed to have all the Views fixed. This approach allowed us for fast updates in the code and good bug traceability since there is less repeated code.

4.2 Technical details about the Product

The client side of the S2B is the downloadable application itself. The initial supported Operating Systems will be Android, with the app being distributed through the official **Google Play** store, with an iOS version of the application coming at a later date if there will be demand for it.

4.2.1 User Interface

The application aims to have a simple UI in order to make it intuitive and easy to read. As a whole, the application is meant to be used in Portrait mode, as it felt more intuitive to navigate and use during initial testing. Initially the application will only feature a Light mode that mainly uses bright colors, with a Dark mode coming in the near future.

4.2.2 Tablet support

The application will provide a slightly different user interface meant to improve the looks and intuitiveness on larger screens, such as tablets.
Since the application is meant for Portrait mode, it will retain the 9:16 standard resolution of smartphones, but certain elements and screens will be either resized to better fit the larger screen or overhauled.
The screens with the bigger difference are the Profile Screen, the Search Skill Screen and the Badges Screen, as shown in Chapter 3.2

4.2.3 Localization

While the prototype only supports English as a language, the plan for the complete S2B is to have the application support additional languages (e.g. French, German, Italian, Chinese, etc.)

4.3 Development and external services

4.3.1 Programming

To implement the application we will mainly use the **Jetpack Compose** framework, an open-source declarative UI framework developed by Google that is mainly used for Android as a way to share UIs across multiple platforms from a single codebase using the Kotlin programming language. Jetpack Compose is designed to integrate seamlessly with existing Android apps and libraries, granting compability with any Android functionality needed.
In Jetpack Compose, functions can be annotated with the *@Composable* annotation, making them composable functions that can be used and combined together to create any UI needed, defining the screen state and actions.
While Jetpack Compose is mainly tailored for native Android development, in the recent year a new version, called **Compose Multiplatform**, was released, adding support for other platforms such as Windows, macOS, Linux and iOS, as such the creation of an additional iOS version of the application would not require to fully start from scratch with other frameworks or programming languages.

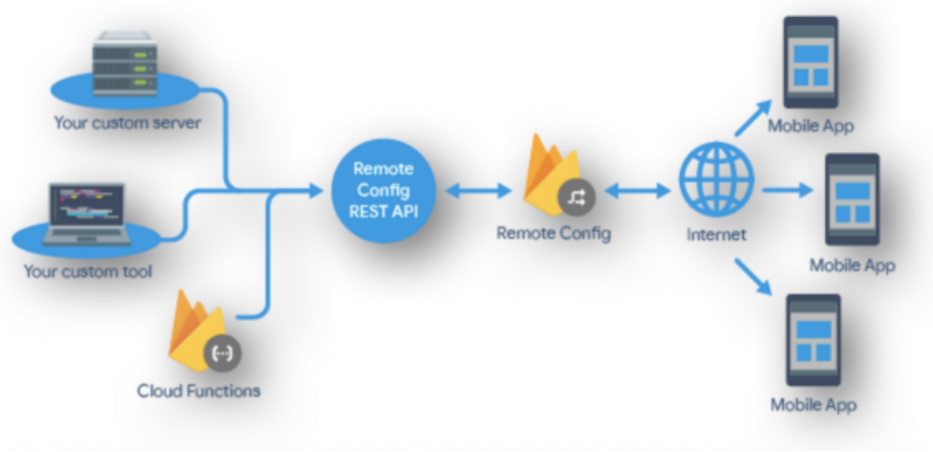
4.3.2 Integration

An extremely helpful tool for the development of the app will be the usage of **git**, which not only allows us to easily share code but thanks to the presence of the *branches* feature allows us to keep a **main** branch where we'll keep the release code while additional features would be first implemented in separated **feature-branches** that will merge to the main once completed and tested.

4.3.3 Firebase Authentication

For user authentication we will rely on the **Firebase** provider, which is natively supported by Jetpack Compose itself.
As shown in the following picture, the authentication handler answers with a token which is given as Authorization header in each back-end HTTP request. At each API call, the back-end check the authenticity of the token together with Firebase.
Additionally, the Firebase SDK automatically handles the token expiration,

requesting a new token when needed.
Firebase supports multiple authentication methods, but for our application the supported ones will be mainly Sign In with Google and a custom Login with eMail and Password



4.4 Testing

Testing was a very important part of the development of the S2B. This project being a mobile application, several aspects had to be taken into account when testing the application.

- We first had to take into account the technical functionality of all the features that are presented in the application.
- Being a user-driven project, we also had to take into account the user experience when using the app, even if completely functional, to take into account how intuitive and enjoyable was the application to users.

4.4.1 Functionality Testing

Thanks to how testing in Jetpack Compose works, we could easily write a series of tests that simulated various scenarios and user actions to check the correct behavior of the application on multiple levels, mainly **Unit Testing** and **Widget Testing**,**Integration testing**. Since it is a
With the testing phase we aim to achieve a code coverage of at least 80% to ensure that the application launches with as few problems as possible.
Additionally, we plan for a final testing phase, the **Acceptance testing**, where the application would be distributed to selected alpha testers through the Google's Play Store alpha channel. During this phase, the testers are

free to use all the functions of the application, checking that it meets the desired criteria and being able to signal any kind of error or problem that might not have been covered by the in-house testing.

4.4.2 Unit Testing

The aim of this phase of testing is to cover the entire business logic of the application itself and to ensure that the user can achieve all of the goals set in the business requirements.
This will be done by providing mocked data and responses to each class and method to ensure that they are correctly handled as expected and the outcome is correct through the use of assertions.
Given the importance of this kind of logic, the goal is to achieve at least a line coverage of 85%, but ideally we should aim to a full 100% coverage.

4.4.3 Widget Testing

The aim of this phase of testing is to ensure that a proper widget is loaded and populated successfully, checking that its UI looks and works as intended on its own by feeding it fictitious data that mocks what is usually received during the usage of the application and making assertions that the various elements are displayed correctly.
Our goal with this testing is to achieve a coverage of at least 75% since not every element can be completely tested on its own and requires the presence of all the elements to be properly displayed.

4.4.4 Integration Testing

As just stated in the previous section, this phase is mainly the subsequent stage of Widget Testing. In this case the aim is to check how the various Widget interact with each other, as such the test consists in starting from a given page and, by simulating the user's action through taps, text insertion and gesture, we assert that the application follows the expected execution flow and that every UI element is properly rendered with the right proportions and functionalities.
For this phase we expect a minimum coverage of 80%.

4.4.5 User experience Testing

User Testing

Throughout the implementation of the application, we used the benefits of the thread-based approach to showcase our application to diverse people of our knowledge. This helped us to establish a cyclic approach:

- We were first developing the functionalities with a small user interface and asking the user if it would feel intuitive to them, if they enjoy it, if they think it is an interesting function to have in the application, and about how they would use it. This aimed at making sure that they understood the functionality, and at understanding the best way to present it to the user.
- Then we would produce the UI that we planned, edited with the ideas that the users gave us to produce a first version of the final UI. We then presented this UI to the user and based on their feedback, we then updated the UI or the functionalities, to be more intuitive, and more enjoyable.

Throughout this process which let the user experience be the center of the development, we obtained ideas about the Reset of the Skills, the History Screen that could be a nice insight, the Profile Screen view was updated multiple times, the Create Screen was reworked also several times to be more intuitive, and the general theme was edited to feel more natural and warmer.

Applying known UX design principles

We also made sure to tackle aspects of known design principles. They are good indicators of important elements that make softwares usable and enjoyable to the users. We mainly used the Nielsen heuristics for this purpose and followed the ones that were explained earlier in the document, in the User Interface part.

Conclusions

This document has described the design and development of the Badge-Up application. Elaborating such a methodology and taking these decisions for the architecture allowed us to have an easier progression, simpler updates, and a fast recovery from bugs.
However, it also allows us to let a huge scalability window, with the possibility to grow the app further. Here are some perspectives that could be developed in the future, from what the user testing phase taught us:

- Making the application available for several languages.
- Adding functionalities in the choice of the Skill registrations.
- Tackling the security and moderation problems that will arise as the application and the community grows.

Overall, these considerations ensure that Badge-Up is well-positioned for continuous improvement and long-term success, while this version can be published and continuously updated with the feedback of the future users.