UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione



TESI DI LAUREA

Progettazione e sviluppo di un'applicazione mobile per il Polimarche Racing Team

Design and development of a mobile application for the Polimarche Racing Team

Relatore

Prof. Emanuele Storti

Candidato

Alessandro Seghini

ANNO ACCADEMICO 2022-2023

Contents

Italian summary **ix**

1 Introduction **1**

2 Requirements analysis **3**

2.1 Team description 4

2.2 Functional requirements 7

2.3 Non-functional requirements 9

2.4 Usability requirements 10

3 Use case analysis **11**

3.1 Use case definition 11

3.2 User roles 12

3.3 Use case diagrams 12

4 Tools **17**

4.1 Flutter 18

4.2 Dart packages 19

4.3 Firebase 21

4.4 Git & GitHub 23

4.5 EmailJS 24

5 Data model **25**

5.1 Firestore collections 26

6 Software development **37**

6.1 Mockups 38

6.1.1 Home 38

6.1.2 Session 40

6.1.3 Setup 42

6.1.4 Problem 44

6.1.5 Telemetry 46

6.2 Navigation schema 48

6.2.1 Member level 49

6.2.2 Foreman level 50

6.2.3 Manager level 51

6.3 Architectural design pattern 52

6.4 Implementation details 54

6.4.1 Login 54

6.4.2 Main page 55

6.4.3 Session page 58

6.4.4 Setup page 64

6.4.5 Problem page 74

6.4.6 Telemetry page 75

6.4.7 New member page 78

7 Database and API using Spring Boot **79**

7.1 ER diagram model of the database 80

7.2 Architectural design pattern 90

7.2.1 Model 92
7.2.2 Repository 94
7.2.3 Service 95
7.2.4 Controller 96
7.3 API development 98
7.3.1 Database connection 99
7.3.2 Model development 99
7.3.3 Repository development 101
7.3.4 Service development 102
7.3.5 Controller development 105
7.3.6 API calls 108
7.4 API test 112
7.4.1 JUnit5 112
7.4.2 Postman 115

8 Conclusion **117**
8.1 Future development 117
8.1.1 Utilization of a dedicated server 117
8.1.2 Concurrency handling in the back-end system 117
8.1.3 Optimization of the database 118

Ringraziamenti **121**

Figures

3.1 Use case diagram - CRU operations for setups 12
3.2 Use case diagram - Displaying and adding sessions in which a driver participated 13
3.3 Use case diagram - Viewing data acquired by sensors for a specific setup during a session 14

5.1 Firestore collections 26
5.2 Firestore collection - Telemetry 36

6.1 Mockup - Home 39
6.2 Mockup - Session 41
6.3 Mockup - Setup 43
6.4 Mockup - Problem 45
6.5 Mockup - Telemetry 47
6.6 Navigation schema - Entry point 48
6.7 Navigation schema - Member level 49
6.8 Navigation schema - Foreman level 50
6.9 Navigation schema - Manager level 51
6.10 Software architecture - Service repository pattern 52

6.11 Login page 54
6.12 Main pages 57
6.13 Session pages 63
6.14 Setup pages 73
6.15 Problem pages 75
6.16 Telemetry pages 77
6.17 New member page 78

7.1 Spring Boot - ER diagram 81
7.2 Spring Boot architecture - Project folder 90
7.3 Spring Boot architecture - Project folder 91
7.4 Test API - Postman 115
7.5 Test API - Database data 116
7.6 Test API - Request result 116

Tables

5.1 Users collection description 27
5.2 Drivers collection description 27
5.3 Agenda collection description 28
5.4 Track collection description 28
5.5 Session collection description 29
5.6 Breakage collection description 29
5.7 Comment collection description 30
5.8 Participation collection description 30
5.9 Wheel collection description 31
5.10 Balance collection description 31
5.11 Spring collection description 32
5.12 Damper collection description 32
5.13 Setup collection description 33
5.14 Used setup collection description 34
5.15 Problem collection description 34
5.16 Occurring problem collection description 35
5.17 Solved problem collection description 35
5.18 Telemetry collection description 36

7.1 API calls 111

Italian summary

L'obiettivo del lavoro è rappresentato dallo sviluppo di un'applicazione mobile che serve a gestire in maniera efficiente l'intero team del Polimarche Racing Team, squadra dell'Università Politecnica delle Marche che compete nel campionato internazionale della Formula SAE.

Il problema principale che ha portato alla realizzazione del software è legato al metodo precedente di gestione delle informazioni del team. Prima, tutte queste informazioni venivano registrate su fogli di carta o documenti Excel separati. Con l'introduzione dell'app, ora è possibile conservare tutte queste informazioni in un unico luogo. Questo rende molto più conveniente e organizzato l'accesso alle informazioni relative a vari setup[1] della monoposto o alle sessioni di test.

L'applicazione è stata prodotta interamente utilizzando **Flutter**, un framework creato da **Google** che mette a disposizione degli sviluppatori un insieme di strumenti utili a realizzare applicazioni di tipo 'cross platform'.

Tra i vari requisiti funzionali, l'applicazione deve fornire la possibilità di distinguere 3 tipi diversi di utenti. Il primo è il manager, che all'interno del Polimarche corrisponde al Team Leader. Il secondo raggruppa tutti i caporeparti, cioè i membri che dirigono i vari reparti del team. Il terzo e ultimo livello raggruppa tutti i

[1] Insieme di parametri della monoposto che permettono di ottimizzare il suo comportamento

restanti membri.

Inoltre, l'app deve consentire al team di eseguire azioni CRU[2] su elementi di maggiore importanza quali setup e sessioni di prova.

In base al livello di autenticazione le funzionalità cambiano. Ad esempio solamente i manager e i caporeparti possono aggiungere e/o modificare un setup o una sessione.

A partire da queste informazioni, è stato creato un piano architetturale per lo sviluppo del software.

Il back-end è stato realizzato in due maniere differenti. Inizialmente si è usato **Firebase**, piattaforma sviluppata da **Google** che mette a disposizione degli sviluppatori un insieme di servizi cloud, quali **Firestore** per gestire i dati e **Fireauth** per l'autenticazione.

A causa della limitatezza di **Firebase**, nel caso in cui in futuro si volessero aumentare le funzionalità dell'app, come per esempio inserire e visualizzare tutti i dati raccolti dai sensori durante una sessione di prova, è stato implementato un programma utilizzando **Spring Boot**, un framework basato su **Spring** che permette di creare un'API. L'applicazione utilizza solo **Firebase**.

Per quanto riguarda il front-end, il software è stato sviluppato in **Flutter**. L'infrastruttura dell'applicazione è rappresentata dal pattern **service repository** che permette di separare la logica aziendale(service), detta anche business logic, dalla gestione dei dati(repository) per promuovere la modularità e la manutenibilità del codice.

[2]Create, Read, Update

Capitolo 1

Introduction

As mentioned in the article 'Mobile application and its global impact'(refer to [1]), "mobile applications are consist of software/set of program that runs on a mobile device and perform certain tasks for the user". The spread of this concept began to take shape in the early 2000s, when phones started to be considered not only as devices allowing users to call someone or send text messages, but as devices that can change completely our life.

Indeed **Java Platform, Micro Edition**, also known as **J2ME**, was built during those years as a platform for developing applications for mobile phones. With the launch of the **Apple Store** in 2008, the transition from non-customazible phones to devices where users could download any application they desired was completed.

Additionally, having an application that helps users schedule daily activities or tasks can be a significant time saver if the software is studied thoroughly .

The aim of this thesis is to efficiently guide the development of a Flutter application through the software development life cycle.

The application was built to meet the needs of the **Polimarche Racing Team**, a group of students from Università Politecnica delle Marche participating in the Formula SAE championship, a collegiate engineering competition that challenges

students to design, build, and compete with small, formula-style racing cars. The competition is open to university and college students from around the world, and it serves as a platform for them to apply their engineering knowledge and skills in a real-world automotive design and manufacturing project.

Here's a summary of all the chapters:

- In ***Chapter 2*** is described the phase in which all the requirements are gathered and analyzed.
- In ***Chapter 3*** are provided and further studied the use cases created from the requirements described in the previous chapter.
- In ***Chapter 4*** are described all the tools used to develop and build the application, starting from **Flutter** framework.
- In ***Chapter 5*** is provided a list in which are visualized the structures of all the data saved in each collection of Firestore.
- In ***Chapter 6*** is summed up the entire software development of the application.
- In ***Chapter 7*** is summed up the entire software development of the API using Spring Boot.

Capitolo 2

Requirements analysis

Requirements analysis serves as a foundation stage for the development of any application. It involves gathering and enumerating the significant and essential activities that the software must accomplish.

To begin it is necessary to collect all pertinent information to understand the needs of various stakeholders, which can include clients, users, managers and many others.

Other than that, requirements should be divided in two different categories. The first one gathers functional requirements, which defines what the software must do. The second one instead involves non functional requirements, which defines the system qualities not related to the functions the software provides.

Furthermore, speaking about requirements, the two categories can be extended to a third. This one gathers usability requirements and it will be covered and explained in section 2.4.

2.1 Team description

To understand accurately the objectives of this project, it is necessary to provide an introduction of the **Polimarche Racing Team**, describing what they do and how they are organized.

As mentioned in the **Introduction**, the team competes in the Formula SAE championship and is composed entirely by students attending the Università Politecnica delle Marche.

Team scope

Their scope is to design and build entirely a car that will be later evaluated by some experts in terms of design, costs, and performance during a series of static and dynamic tests.

Before these evaluations, the car undergoes testing during several sessions organized by the team. There are four different types of sessions: **Endurance**, **Acceleration**, **Autocross** and **Skidpad**.

Once they define a new session, they also lay down the drivers that will test the car, along with the order in which each of them should drive.

Since in some sessions the car may face breakages, they also list down all of them with a small description and noting if the breakage was caused by the driver or not.

In all of them, they test primarily how different parameters affect the performance of the car. In other words, the team initially define a setup, that can be later modified, for each session.

A setup is characterized by:

- Four sets of **wheels**. Each wheel is defined by the following information:

  - *codification* → defines general information of the wheel;
  - *position* → defines the position of the car in which the wheel is used(e.g., Front right);
  - *pressure* → defines the pressure of the wheel;
  - *camber* → defines the frontal inclination of the wheel;
  - *toe* → defines the upper inclination of the wheel;

- Two sets of parameters for car's **balance**, comprising weight and brake percentage for both front and rear end of the car.

- Two couples of **springs** for both front and rear end of the car. Each couple describes the following information:
  - *codification* → defines general information of the springs;
  - *position* → defines the position of the car in which the springs are used(e.g., Front);
  - *height* → defines the height of the springs from the wheel;
  - *arb position* → defines the position of the ARB[1];
  - *arb stiffness* → defines the stifness of the ARB;

- Two couples of **dampers** for both front and rear end of the car. Each couple describes the following information:
  - *position* → defines the position of the car in which the dampers are used(e.g., Front);
  - *lsr* → defines the low speed rebound;

[1] ARB stands for Anti-Roll Bar. It is a suspension component used in many vehicles to reduce body roll during cornering.

  - *hsr* → defines the high speed rebound;
  - *lsc* → defines the low speed compression;
  - *hsc* → defines the high speed compression;

- The position of the **front wing hole**. This parameter is related to the aerodynamics of the car.

The team often writes down some notes for a specific setup to describe the car's behaviour with that setup during the test.

Additionally, the team gathers telemetry data from sensors positioned into the car while they test it out. Those data are then studied to increase the performance of the car.

They also save all the problems the car faced. For each of them, they also lay down all the setups in which the problem is currently occurring, and in which the problem was already solved.

Team organization

The team is organized in a pyramid-like structure, with **managers** at the top. There are three different managers within the team.

Managers are the members that make decisions and manage the entire team. They're responsible for the creation of setups and sessions, as well as the admission of new students to the team.

The team is then divided in multiple departments, and each one has a precise goal. To oversee the work done within, a new figure was depicted by **foremen**. Each department is lead by a single foreman.

The base of the pyramid gathers all the other **members** who are divided into departments.

2.2 Functional requirements

From the introduction of the team done in section 2.1, it can be defined a list of all the functional requirements.

FR 1

"Managers are the members that make decisions and manage the entire team. They're responsible for the creation of setups and sessions, as well as the admission of new students to the team.
They also save all the problems the car faced (problems)."
The application must provide pages allowing managers and foremen to execute CRU[2] operations for setups, practice sessions and also problems the car may face.

FR 2

"Since in some sessions the car may face breakages, they also list down all of them with a small description and noting if the breakage was caused by the driver or not."
The application must allow managers and foremen to save records of car breakdowns along with their causes for each session.

FR 3

"Managers are responsible for the admission of new students to the team."
The application must allow managers to add new students to the team.

[2]CRU stands for create, read and update

FR 4

"Once they define a new session, they also lay down the drivers that will test the car, along with the order in which each of them should drive."

The application must allow managers and foremen to add and display all the drivers that drove the car during a session.

FR 5

"Additionally, the team gathers telemetry data from sensors positioned into the car while they test it out. Those data are then studied to increase the performance of the car."

The application should allow users to view data acquired by sensors for a specific setup during a session.

FR 6

"The base of the pyramid gathers all the other members who are divided into departments."

The application should provide a page in which users can view all members of the team ordered by workshop department.

FR 7

"There are three different managers within the team."

The application must define three login levels comprising managers, foremen and members.

2.3 Non-functional requirements

Listed below are some non-functional requirements.

NFR 1

The application must use a DBMS[3] to store and retrieve data pertinent to the application.

NFR 2

The development of the application should be accomplished using Flutter framework and Dart programming language.

[3]A DBMS is a software used to store, retrieve, and run queries on data

2.4 Usability requirements

Usability is a measure of how well a specific user in a specific context can use a product to achieve a defined goal effectively, efficiently and satisfactorily. Based on this definition, the design of any mobile application should incorporate three essential elements.

The first is effectiveness, that supports users in completing actions accurately.

The second is efficiency, that allows the user to perform tasks quickly.

The third is engagement, which measures the user's level of satisfaction during the use of the application.

Hence, there is a need to add other requirements specific for usability concerns.

For a more comprehensive understanding of these requirements, refer to the article [2].

Capitolo 3

Use case analysis

Use case analysis's goal is to identify roles and use cases of the software throughout a transformation of all the requirements gathered during the previous phase.

3.1 Use case definition

A use case defines the actions that can be achieved by performing a series of steps by a user role, which is also called actor.

In this case, an actor may represent a physical person or even an abstract system.

3.2 User roles

Thanks to the preceding functional requirements, the user roles can be established. They are as follows:

1. Managers
2. Foremen
3. Members

3.3 Use case diagrams

This section will showcase use case diagrams that involve some of the functional requirements.

CRU operations for setups

Since all CRU operations are basically the same, the diagram is reported only once for all the functional requirements concerning CRU.

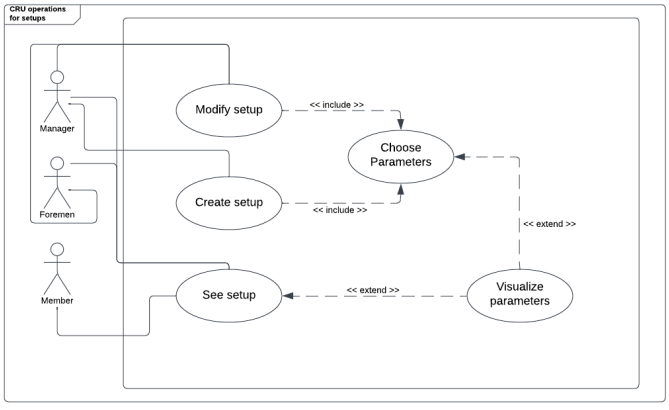


Figura 3.1: Use case diagram - CRU operations for setups

As shown by Figure 3.1, managers and foremen are the ones that have the privilege to create and modify setups. These two features are represented as use cases, both of which include another use case called **Choose Parameters**. It allows the corresponding actors to select all the parameters that will characterize the setup.

Choose Parameters is further extended by another use case called **Visualize Parameters**, which is responsible for the visualization of all the parameters that previously used in other setups. Additionally, it also extends the use case **See Setup**, representing a common functionality for all users, allowing them to view all the saved setups.

Displaying and adding sessions in which a driver participated

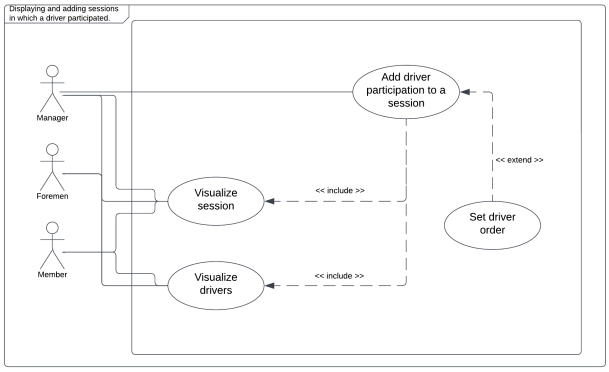


Figura 3.2: Use case diagram - Displaying and adding sessions in which a driver participated

As shown by Figure 3.2, managers have the privilege of adding a driver participation to a session. The corresponding use case includes two different use cases and they're related functionalities are in common for all users.

The first one is **Visualize Session**, which allows users to view all the sessions saved, and the second one is **Visualize Drivers**, which allows users to see all the drivers of the team.

The last use case, **Set Driver Order**, is responsible for establishing the correct order in which drivers test the car during a session.

Viewing data acquired by sensors for a specific setup during a session

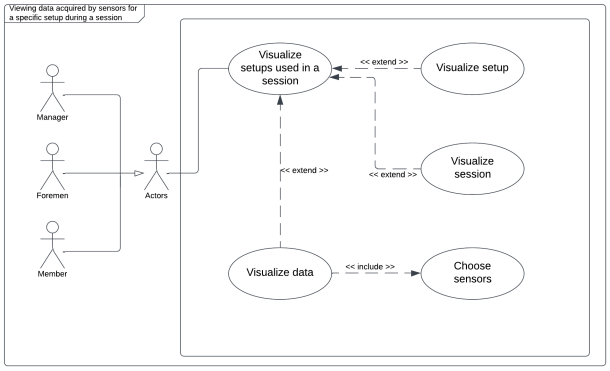


Figura 3.3: Use case diagram - Viewing data acquired by sensors for a specific setup during a session

As shown by Figure 3.3, unlike all the previous use cases diagrams, all actors converge into the same figure.

The use case **Visualize setups used in a session** is related to the functionality that allows users to view all the different setups that were tested during a session. It extends three different use cases.

The first one is **Visualize Setup**, and it allows users to view all the setups saved.

The second one is **Visualize Session**, and it allows users to see all the sessions saved.

The third one is **Visualize Data**, and it allows users to view the data gathered from all the sensors used. It includes the use case **Choose Sensors**, and its corresponding functionality allows users to change the type of data to be visualized within a specialized chart.

Capitolo 4

Tools

To develop a good application, it's compulsory to proficiently utilize a list of tools, keeping in mind that each one should have a specific purpose.

In this chapter are listed and described all of the instruments employed, ranging from Flutter framework to the version control system known as Git.

4.1 Flutter

Flutter [3] is a cross-platform framework brought into life by Google utilized to create a wide range of software applications. These applications, written in **Dart** [4], span from web to mobile applications.

Differently from many other frameworks, the code of any application needs to be written once only, allowing it to function seamlessly across multiple platforms. This unique characteristic makes Flutter a single codebase[1] platform.

The central idea behind Flutter is the use of widgets, which, when combined together, define the user interface. They can represent structural elements like buttons or texts, as well as stylistic elements like colors, and many others.

Another important feature regards the possibility to extend the capabilities of apps. The Dart language offers a repository full of different packages that can be installed in an application by simply adding a line of code within a configuration file called **'pubspec.yaml'** .

[1] A codebase, or code base, is the complete body of source code for a software program.

4.2 Dart packages

As specified in the previous section, a package is used to add functionality to a Flutter application.

Here's a list of all the packages used within the application.

```
cupertino_icons: ^1.0.2
  // iOS high-fidelity widgets
google_nav_bar: ^5.0.6
  // google style nav bar
flutter_inset_box_shadow: ^1.0.8
  // inset property to BoxShadow
flutter_svg: ^0.22.0
  // render SVG assets
fluttertoast: ^8.0.8
  // create toast messages
table_calendar: ^3.0.9
  // highly customizable calendar widget
step_progress_indicator: ^1.0.2
  // bar indicator made of a series of selected
  // and unselected steps
hidden_drawer_menu: ^3.0.1
  // add a drawer mode menu feature with perspective
  // animation
firebase_core: ^2.15.1
  // enable the use of Firebase Core API
firebase_auth: ^4.8.0
  // enable the use of Firebase Authentication API
```

```
cloud_firestore: ^4.9.0
  // enable the use of Cloud Firestore API
provider: ^6.0.5
  // mechanism allowing to write inherited widgets
  // with ease
uuid: ^4.0.0
  // create ids
syncfusion_flutter_charts: ^22.2.11
  // enable use of charts
http: ^0.13.3
  // manage http request
```

4.3 Firebase

Firebase [5] is a combination of back-end cloud computing services, such as cloud storage using a NoSQL database, authentication, real time database and many other features.

The software developed utilizes mainly the cloud storage and authentication components.

As mentioned in the previous section (4.2), to integrate Firebase into a Flutter application, it's required to add a few packages.

The first one is `firebase_core` and it's responsible for connecting your Flutter app to your Firebase project. To ensure the application works correctly using Firebase, the main function should resemble the following.

```
Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  runApp(
    FutureBuilder(
      future: _initializeFirebase(),
      builder: (context, snapshot) {
        if (snapshot.connectionState
                == ConnectionState.done){
          return AppWidget();
        }
        return const Center(
          child: CircularProgressIndicator(),
        );
      }
    );
```

```
    );
}
```

The second one is `firebase_auth` and represents the base for all of Firebase Authentication features.
It provides several methods that will help manage all the authentication methods for an application, such as `signInWithEmailAndPassword` for signing in with an email and password, and `createUserWithEmailAndPassword` for creating a new user with the credentials provided as argument.

The third one is `cloud_firestore` and integrates the possibility to link an application to the firebase cloud storage. This package is used to manage the data saved inside the storage. Unlike traditional MySQL databases, the entire cloud is organized in collections fragmented in documents, each containing data that can have different structure. Additionally, each document can include sub-documents, and these sub-documents can also be further divided.

4.4 Git & GitHub

Git and GitHub [6] are respectively the instruments used for software's version control and web hosting service of Git repositories. Let's analyze them individually.

Git is a source code management tool that tracks changes in the source code and enable developers to work simultaneously on a project. One of the reasons that initially made Git so famous and used is its support for non-linear workflows. Indeed, for every project it's possible to create thousands of parallel branches on different computers, and these branches can be later merged into the main branch, usually known as 'master'.

The most useful git commands are:

- `git clone` → creates a copy of a Git repository on a local machine
- `git commit` → commit a snapshot of the source code into the project history
- `git pull` → fetch a branch from a Git repository and merges it into the current branch
- `git push` → push all the commits to a branch of a Git repository

GitHub instead is a platform and cloud-based service that allows developers to store and manage their code in repositories utilizing Git as the version control system.

4.5 EmailJS

EmailsJS is a back-end service used to send emails automatically defining first their templates.

This tool is an online application that can be accessed trough the following link: `https://www.emailjs.com/`.

Once the login is complete, users can outline the email structure by clicking on `Email Templates` from the menu on the left side of the screen. To use EmailJS effectively, it's important to specify the service that will send the email.

Indeed, this tool allows users to connect to any email service, such as Gmail and Outlook, to send messages automatically with success.

Inside the templates, the values sent from the `Http request` can be utilized to define the body of the email by placing their field names inside double curly braces .

Capitolo 5

Data model

After the introduction of all the tools used to develop the software, it's time to define how data are modelled, stored and collected using Firestore.

Since an approach using concepts taken from relation databases, conveying entities and relations, is easier to follow at first, an E/R diagram was designed.

5.1 Firestore collections

This section provides all the different Firestore collections that were used to develop the application with an additional analysis on the data types within the documents.

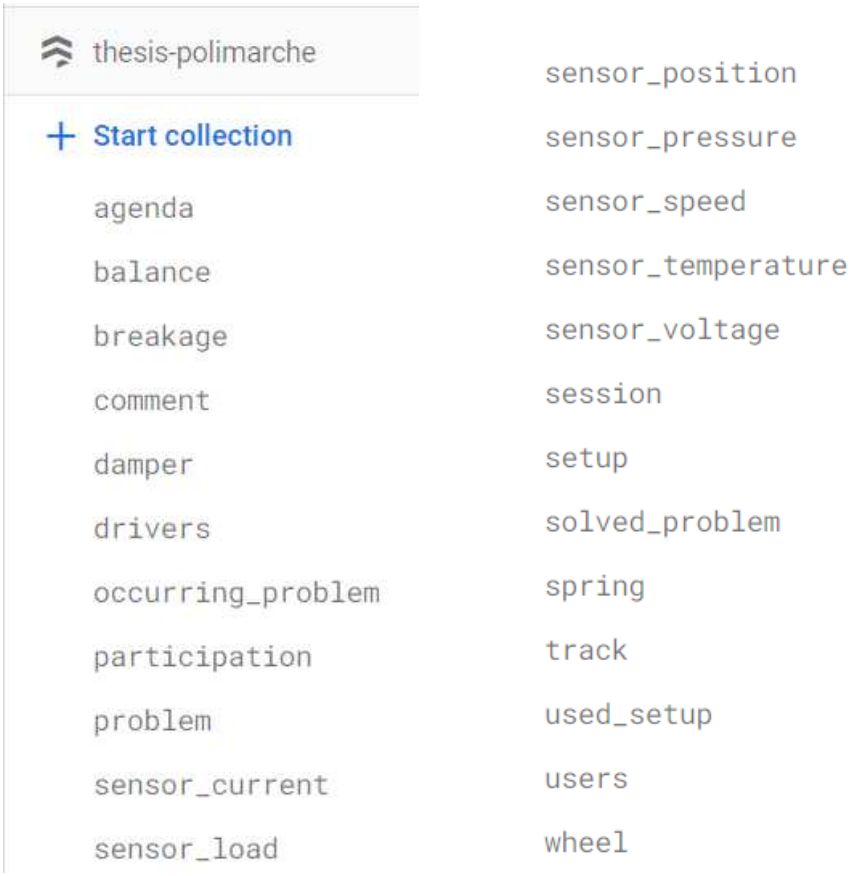


Figura 5.1: Firestore collections

Users

Contains all the members of the team.

Field	Data type	Description
Uid	String	Member's uid when first created in FireAuth
Nome	String	First name of an actual member
Cognome	String	Last name of an actual member
DOB	Timestamp	Date of birth of an actual member
Matricola	Number	Member id within the university
Email	String	Member university email
Telefono	Number	Phone number of an actual member
Ruolo	String	Member's role within the team
Reparto	String	Member's department

Tabella 5.1: Users collection description

Drivers

Contains all the drivers of the team.

Field	Data type	Description
Membro	Map	Member saved in 'users' collections
Altezza	Number	Member's height
Peso	Number	Member's weight

Tabella 5.2: Drivers collection description

Agenda

Gathers all the events for every member.

Field	Data type	Description
Matricola	Number	Member's university id
Data	Timestamp	Date of the event
Inizio	String	Starting time of the event
Fine	String	Ending time of the event
Descrizione	String	Event description

Tabella 5.3: Agenda collection description

Track

Contains all the tracks in which the car can be tried out.

Field	Data type	Description
Nome	String	Track name
Lunghezza	Number	Track length

Tabella 5.4: Track collection description

Session

Contains all the scheduled sessions in which the team tried out the car.

Field	Data type	Description
Evento	String	Type of test tried
Data	Timestamp	Date in which the session was tried
Inizio	String	Starting time of the session
Fine	String	Ending time of the session
Tracciato	Map	Specify the track of the test
Meteo	String	Weather during the session
Pressione atmosferica	Number	Ambiental pressure
Temperatura aria	Number	Air temperature
Condizione tracciato	String	Track condition
Temperatura tracciato	Number	Track temperature

Tabella 5.5: Session collection description

Breakage

Contains all the breakages the car faced during tests.

Field	Data type	Description
Session id	Number	Uid of a session document
Colpa pilota	Boolean	Determine if the breakage was caused by driver fault
Descrizione	String	Small description of the breakage

Tabella 5.6: Breakage collection description

Comment

Contains all the comments done for a specific session.

Field	Data type	Description
Uid	String	Comment identifier
Session id	Number	Uid of a session document
Flag	String	Team equip that added the comment. Can be Team or Pilota
Descrizione	String	Actual comment

Tabella 5.7: Comment collection description

Participation

Contains all the driver participation during tests.

Field	Data type	Description
Session id	Number	Uid of a session document
Matricola pilota	Number	Member's university id
Ordine	Number	Driver order during the test
Cambio pilota	String	Time lost for driver change

Tabella 5.8: Participation collection description

Wheel

Gathers all the combinations of parameters used for wheels.

Field	Data type	Description
Id	Number	Wheel identifier
Codifica	String	Wheel codification
Posizione	String	Wheel position. Can be Ant dx, Ant sx, Post dx or Post sx
Pressione	Number	Wheel pressure
Frontale	String	Specify wheel camber
Superiore	String	Specify wheel toe

Tabella 5.9: Wheel collection description

Balance

Gathers all the combinations of parameters used for balance.

Field	Data type	Description
Id	Number	Balance identifier
Posizione	String	Balance position. Can be Ant or Post
Frenata	Number	Brake balance of the car
Peso	Number	Weight balance of the car

Tabella 5.10: Balance collection description

Spring

Gathers all the combinations of parameters used for springs.

Field	Data type	Description
Id	Number	Spring identifier
Codifica	String	Spring codification
Posizione	String	Spring position. Can be Ant or Post
Altezza	Number	Spring height from the wheels
Posizione arb	String	Drive shaft position
Rigidezza arb	String	Drive shaft stiffness

Tabella 5.11: Spring collection description

Damper

Gathers all the combinations of parameters used for dampers.

Field	Data type	Description
Id	Number	Damper identifier
Posizione	String	Spring position. Can be Ant or Post
Lsr	Number	Low speed rebound
Hsr	Number	High speed rebound
Lsc	Number	Low speed compression
Hsc	Number	High speed compression

Tabella 5.12: Damper collection description

Setup

Contains all the combinations of parameters used to optimize the behaviour of the car.

Field	Data type	Description
Id	Number	Setup identifier
Ala	String	Front wing
Note	String	Notes of a setup
Wheel ant dx	Number	Wheel ant dx parameters used
Wheel ant sx	Number	Wheel ant sx parameters used
Wheel post dx	Number	Wheel post dx parameters used
Wheel post sx	Number	Wheel post sx parameters used
Balance ant	Number	Balance ant parameters used
Balance post	Number	Balance post parameters used
Spring ant	Number	Spring ant parameters used
Spring post	Number	Spring post parameters used
Damper ant	Number	Damper ant parameters used
Damper post	Number	Damper post parameters used

Tabella 5.13: Setup collection description

Used setup

Gathers all the setup used during a session.

Field	Data type	Description
Session id	Number	Id of the session in which the setup was used
Setup id	Number	Id of the setup used
Commento	String	Comment for the setup during its use

Tabella 5.14: Used setup collection description

Problem

Contains all the problems experienced from the car.

Field	Data type	Description
Id	Number	Problem identifier
Descrizione	String	Problem description

Tabella 5.15: Problem collection description

Occurring problem

Gathers all the problems occurring in all the setups.

Field	Data type	Description
Id	Number	Occurring problem identifier
Descrizione	String	Occurring problem description
Setup id	Number	Id of the setup in which the problem occur
Problem id	Number	Id of the problem occurring in the setup

Tabella 5.16: Occurring problem collection description

Solved problem

Gathers all the problems solved in all the setups.

Field	Data type	Description
Id	Number	Occurring problem identifier
Descrizione	String	Occurring problem description
Setup id	Number	Id of the setup in which the problem was solved
Problem id	Number	Id of the problem solved in the setup

Tabella 5.17: Solved problem collection description

Telemetry

Since there are multiple collections regarding data obtained from the sensors, and all of them gather documents with a similar structure, telemetry congregates all the collections grouped inside the black border.

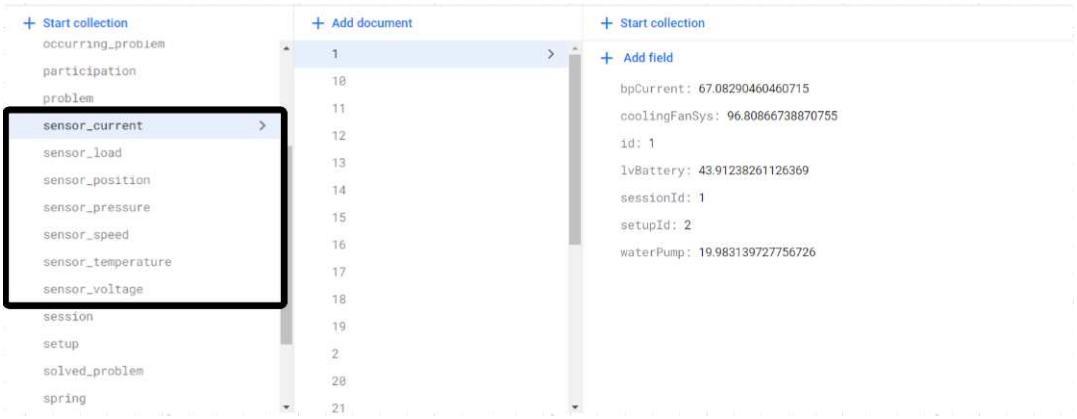


Figura 5.2: Firestore collection - Telemetry

Field	Data type	Description
Id	Number	Occurring problem identifier
Setup id	Number	Id of the setup on which the data was gathered
Session id	Number	Id of the session in which the data was gathered
...	Number	...

Tabella 5.18: Telemetry collection description

Capitolo 6

Software development

This phase of the software development life cycle mainly regards code production and the techniques used to bring the software to life. For this reason the implementation process takes huge importance and it's probably the lengthiest step to fulfill entirely.

This chapter will provide and explain both the code and the architecture used, as well as general information such as mockups and navigation schemas.

6.1 Mockups

6.1.1 Home

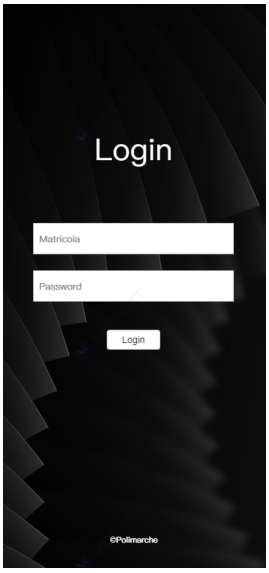
The first page of the application is represent by Figure 6.1a and is used as the login page. Once users enter their credentials correctly, they will be redirected to the home page described in Figure 6.1b.

From this page, users can choose to navigate to other pages using the bottom navigation bar or the buttons within the layout.

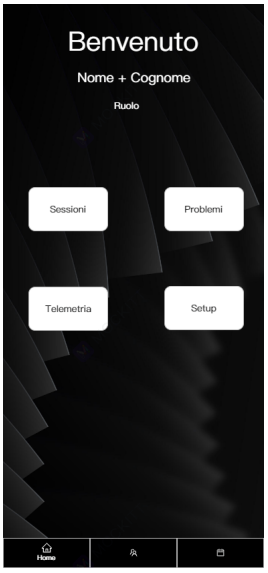
The navigation bar redirects user to the pages depicted in Figure 6.1c and 6.1d.

The first page mentioned allows users to view all the members of the team. Each member is represented by a card in which a button is placed. When clicked, this button redirects users to pages 6.1e and 6.1f, respectively, depending on whether the member is a simple member or a driver.

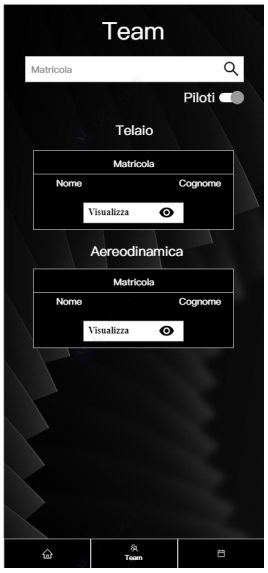
Instead, the buttons redirect to a set of pages specific for the completion of tasks related to a field of the team. All the pages are described in the next sections.



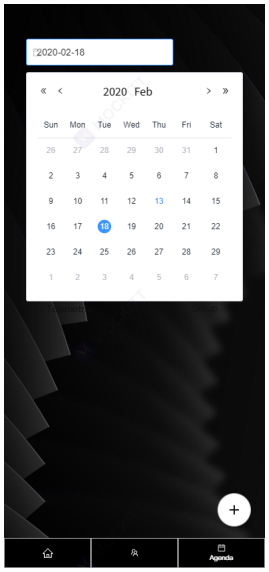
(a) Login



(b) Home



(c) Team



(d) Agenda



(e) Member



(f) Driver

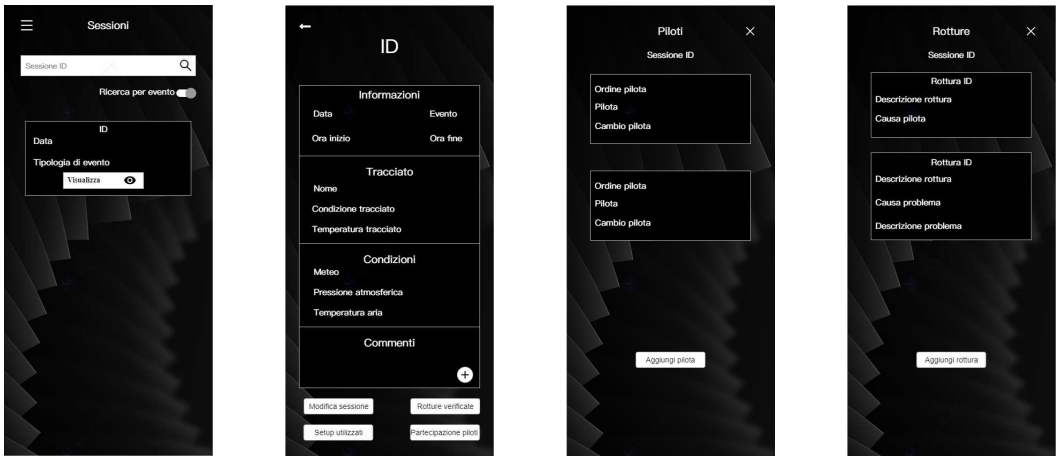
Figura 6.1: Mockup - Home

6.1.2 Session

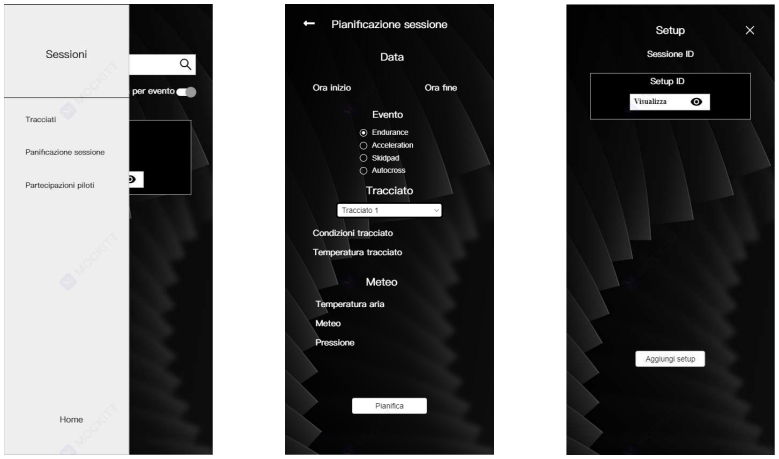
The first page that opens up when users click on session button on the home page of the application is depicted in Figure 6.2a. This page contains all the planned sessions and a button in the top-left of the screen. When clicked, it opens the drawer described in Figure 6.2e, from which users can plan a session or see the list of all the tracks.

Each session is displayed as a card having a button that users can click to see the details of the selected session. The page has the same layout of the Figure 6.2b.

The details page of a session contains multiple buttons displayed in the bottom part of the screen and they allow users to perform other tasks, such as visualizing all the drivers who participated in the selected session and all the breakages happened, as shown in Figure 6.2c and 6.2d.



(a) Main (b) Session (c) Participation (d) Breakages



(e) Drawer (f) Plan (g) Setup

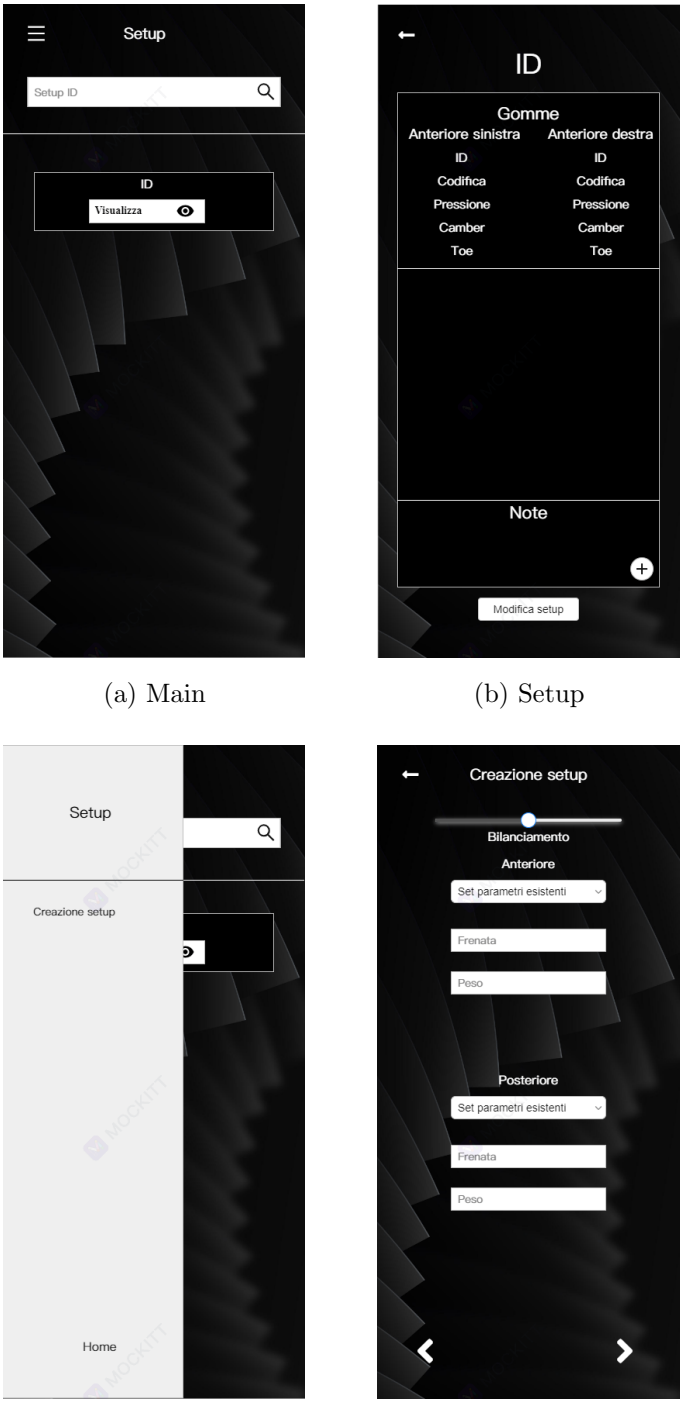
Figura 6.2: Mockup - Session

6.1.3 Setup

The first page that opens up when users click on setup button on the home page of the application is depicted in Figure 6.3a. This page contains all the saved setups and a button in the top-left of the screen. When clicked, it opens the drawer described in Figure 6.3c, from which users can create a new setup.

Each setup is displayed as a card having a button that users can click to see the details of the selected setup. The page has the same layout of the Figure 6.3b.

The modification and the creation page of a new setup follows the layout described in Figure 6.3d.



(a) Main (b) Setup

(c) Drawer (d) Plan

Figura 6.3: Mockup - Setup

6.1.4 Problem

The first page that opens up when users click on problem button on the home page of the application is depicted in Figure 6.4a. This page displays a list of all the problems found in all the setups. Each problem is presented as a card with a button. Clicking on this button redirects users to the page described in Figure 6.4b.

The layout of this page consists of a list of cards, each one depicting a setup in which the problem occurs. Additionally, there is a bottom navigation bar that allows users to navigate to page 6.4c, which lists all the setups where the problem has already been solved.



(a) Main



(b) Occurring problems



(c) Solved problems

Figura 6.4: Mockup - Problem

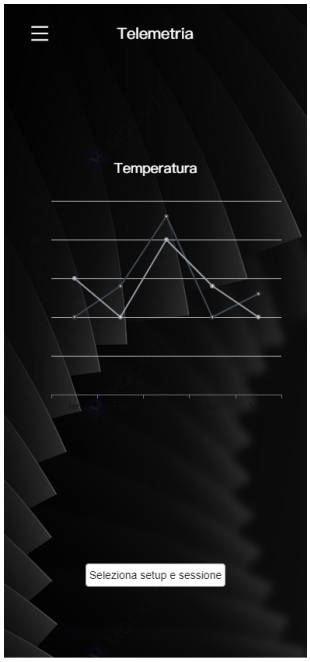
6.1.5 Telemetry

The first page that opens up when users click on telemetry button on the home page of the application is depicted in Figure 6.5a. This page provides two different dropdown menus used to select the session and the setup for which users want to view the telemetry data.

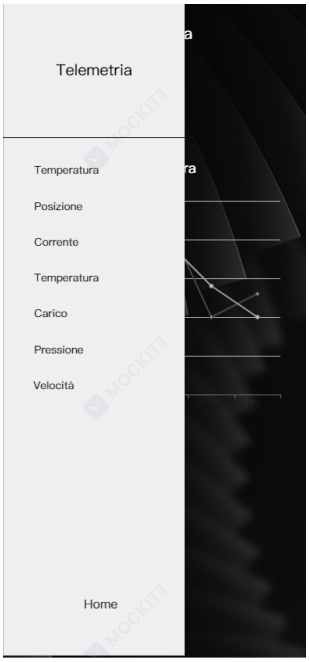
Once users have selected these values, they will be redirected to the page described in Figure 6.5b. The layout includes a chart displaying the data and a drawer menu, as shown in Figure 6.5c. In this drawer menu, users can choose the type of data they wish to visualize.



(a) Main



(b) Charts



(c) Drawer

Figura 6.5: Mockup - Telemetry

6.2 Navigation schema

A navigation schema defines how an user can move between different pages or views within an application whether it's a mobile app, web app, or any other type of software.

As written in the last functional requirement (refer to paragraph 2.2), which states there must be three different login levels, for every authentication level is designed a navigation schema. Initially, all users begin on the same page.

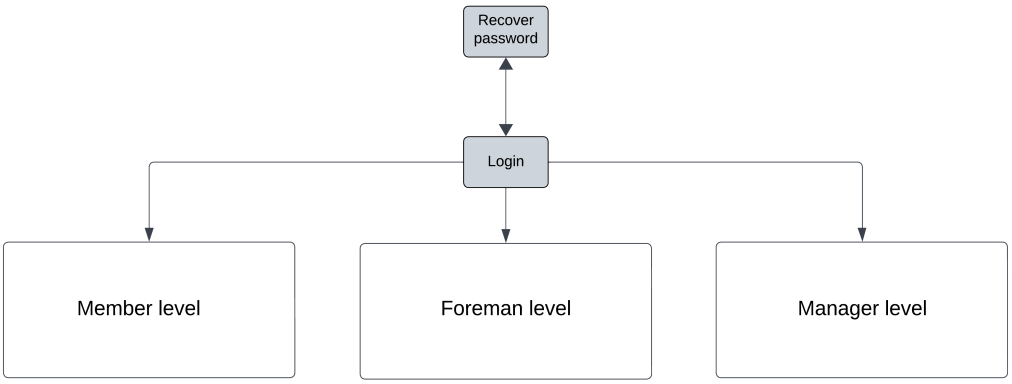


Figura 6.6: Navigation schema - Entry point

From now on, the navigation schema will refer to each authentication level.

6.2.1 Member level

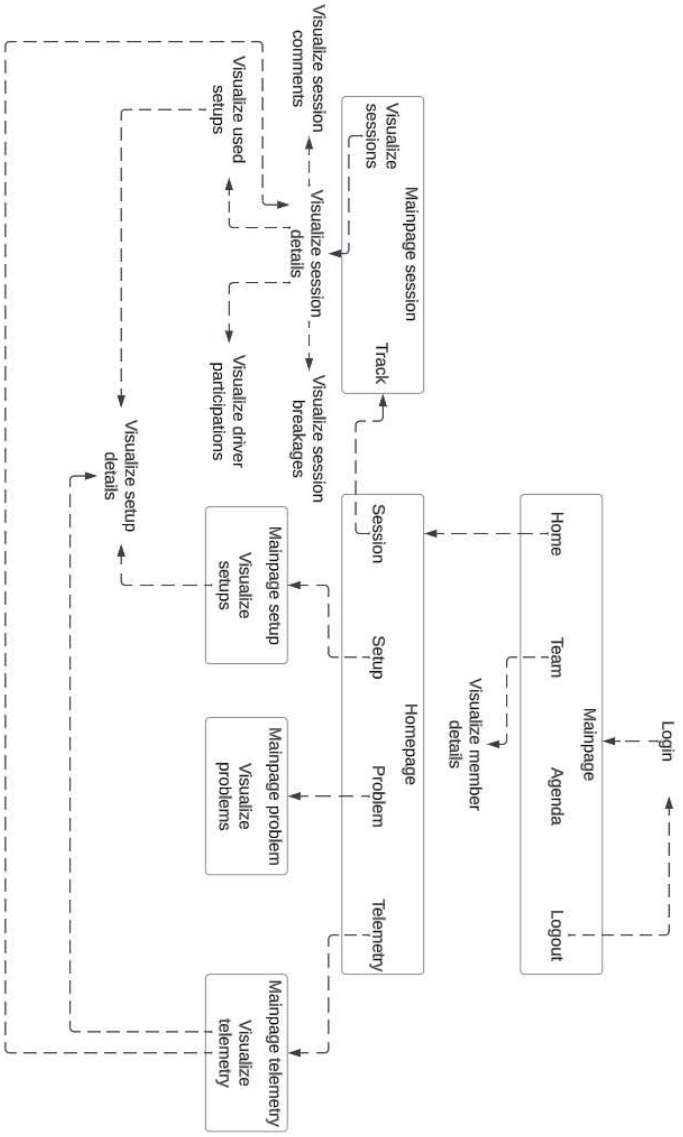


Figura 6.7: Navigation schema - Member level

6.2.2 Foreman level

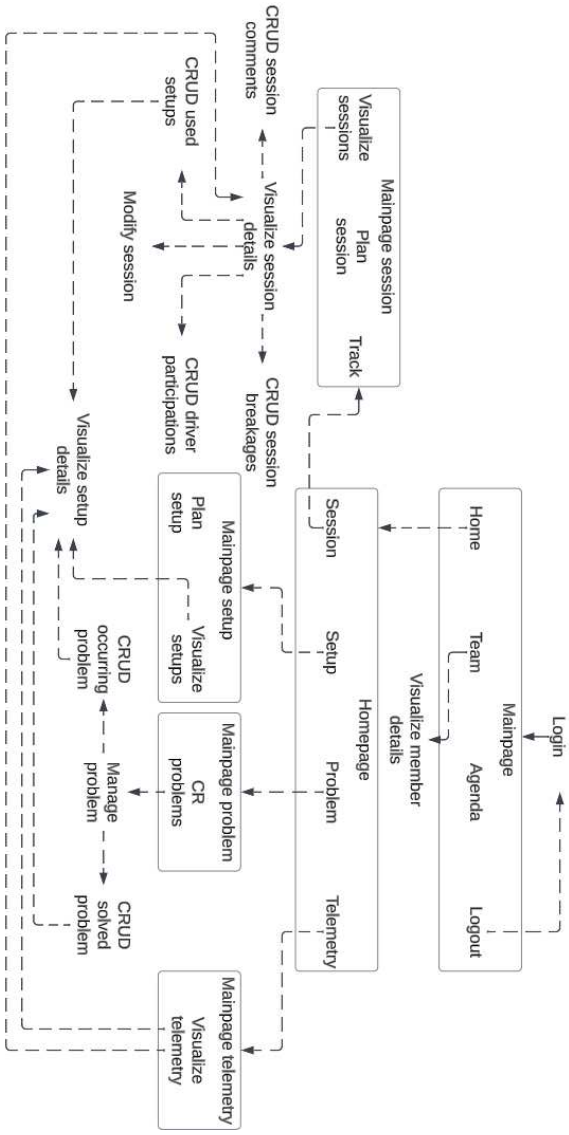


Figura 6.8: Navigation schema - Foreman level

6.2.3 Manager level

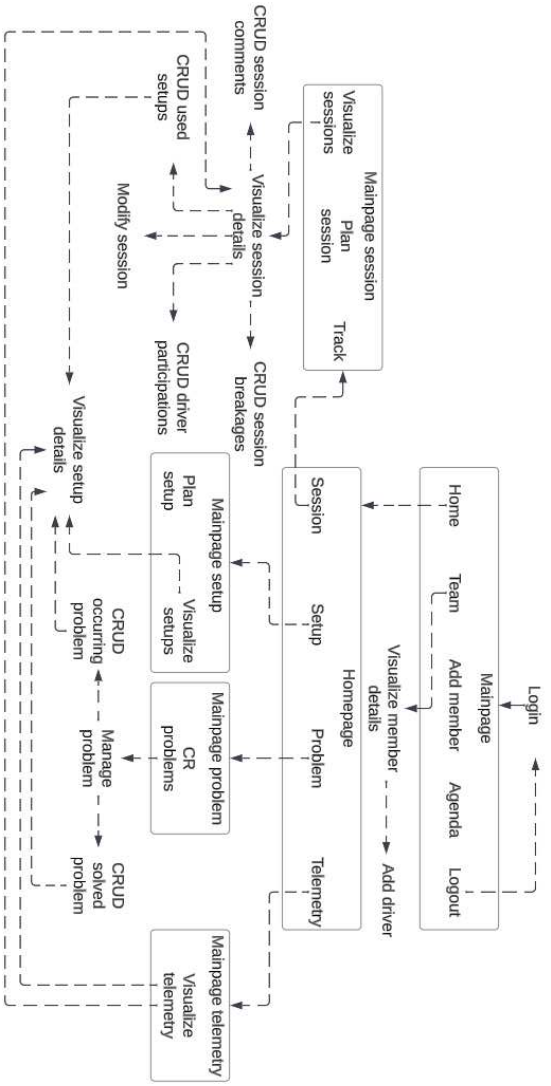


Figura 6.9: Navigation schema - Manager level

6.3 Architectural design pattern

The pattern chosen to implement the entire application is the **service repository pattern**.

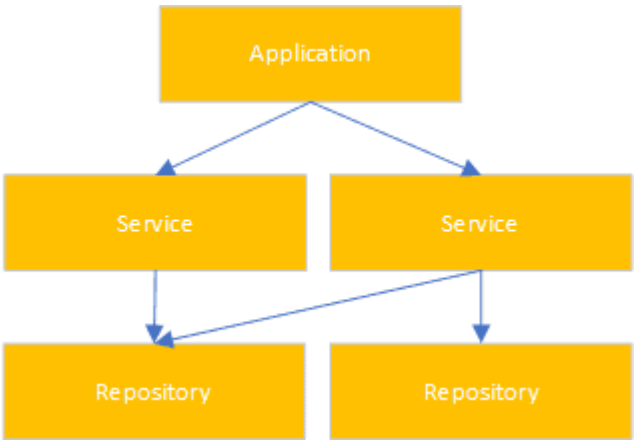


Figura 6.10: Software architecture - Service repository pattern

The idea of this pattern is that your code, displayed as the **application layer** in the figure above, should be completely unaware of the business logic and the data layer.

It generally works in conjunction with a simple and basic repository pattern to achieve a greater separation of concerns. This means that all the properties of the repository pattern is inherited by the pattern chosen.

In software engineering a repository is a shared database about engineered artifacts, including documents, information systems and many others [7].

Indeed, in the context of the service repository pattern chosen, a repository takes on a slightly different role. Its primary purpose is to isolate the data layer from the rest of the application which contributes to the separation of concerns. Furthermore, repositories are responsible for mapping the data retrieved from the

storage format to the business objects specified inside the **model layer** of the application.

A service is instead a module that stands in the middle of the application and data layer. It is generally used to send requests to the repositories for CRUD operations, and, in case a return is expected, it will pass the data back to the application layer.

All model classes include two methods. The first is toMap(), which maps an object of the class into a data structure having a consistent format for the database. The second one is fromMap(), and it serves as the counterpart to the toMap() method. Indeed, it is used to map the data retrieved from the data source into an object designed accordingly to the model class.

6.4 Implementation details

Since the manager authentication level is the one in which all functions and operations provided by the application converge.

6.4.1 Login

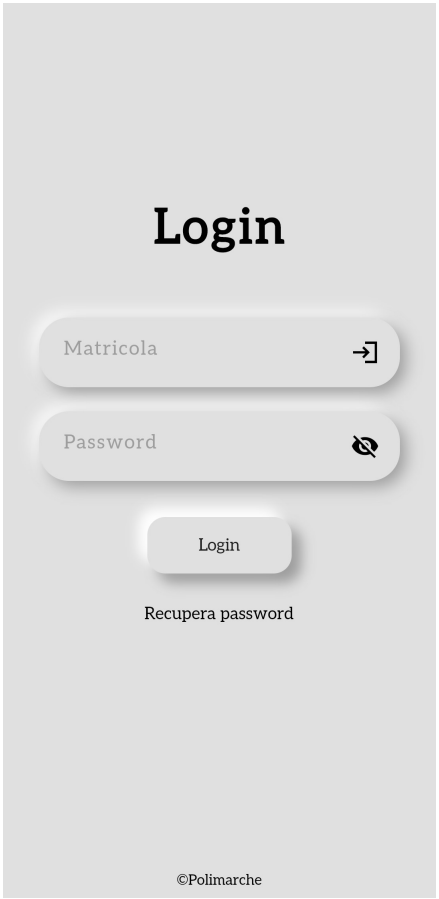


Figura 6.11: Login page

The login page provides a login form and a password recovery method.

If users wish to recover their password, they should click on the text located right under the login button. In the dialog that pops up, they should fill the

'matricola'[1] field. If, within Fireauth[2], there is already a user registered with the university email, which follows the format 'S + student ID number + studenti.univpm.it', Firebase will handle the recovery automatically by sending an email to the address associated with the student ID number.

6.4.2 Main page

Once the manager logs in, inside the main page there will be three different layouts.

The first layout, as shown in Figure 6.12a, is a simple homepage with some buttons that redirect the user to other pages.

The second layout, as shown in Figure 6.12b, lists all the team members, categorized under the workshop area they belong to. Each member is accompanied by a small button that enables users to check their details. Clicking this button opens a new page, which can have two different layouts. If the member is not a driver, managers can assign them as a driver. If the member is already a driver the layout contains additional information such as height and weight. To view all the drivers, users can click on the car image positioned at the top of the list. Attached is a snapshot of the code used to correctly display all the members.

```
// called whenever the input inside the search bar changes
void filterListByQuery(String query) {
  if (query.isNotEmpty && !isDriverPressed) {
  // display members
    setState(() {
      _filteredTeamList = _teamMembers
          .where((element) =>
            element is String ||
```

[1]Matricola is the university student ID

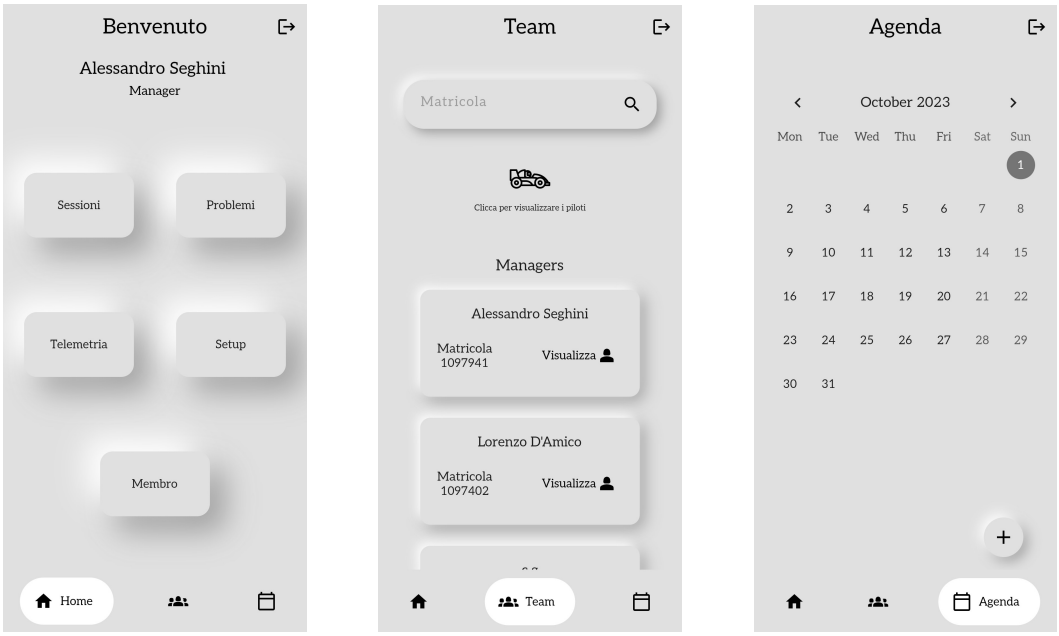
[2]Firebase Authentication

```
            (element is Member &&
              element.matricola.toString().contains(
                query)
            )
          ).toList();
    });
  } else if (query.isNotEmpty && isDriverPressed) {
  // display drivers
    setState(() {
      _filteredTeamList = _teamDrivers
        .where((element) =>
          element is String ||
          (element is Driver &&
            element.membro.matricola.toString().contains(
              query)
          )
        ).toList();
    });
  } else if (query.isEmpty && isDriverPressed) {
  // display drivers
    setState(() {
      _filteredTeamList = _teamDrivers;
    });
  } else {
  // display members
    setState(() {
      _filteredTeamList = _teamMembers;
```

```
      });
    }
}
```

The list `_filteredTeamList` is used by the `ListView.builder` to display the members. It is defined as a `List<dynamic>` because it can contain different types of data. `_teamMembers` and `_teamDrivers` store all the members and drivers retrieved from the data source.

The third and last layout, as shown in Figure 6.12c, contains a personalized agenda, where users can view, add, modify and delete events.



(a) Home (b) Team (c) Agenda

Figura 6.12: Main pages

6.4.3 Session page

By clicking on the top-left button within the homepage, users will be redirected to a new page where they can manage sessions.

In the app bar there's an icon that, when clicked, opens a menu[3] with three different entries. The menu is depicted in Figure 6.13b.

The first entry redirects users to the main page of the session layout, where all the sessions saved are listed.

The second entry redirects managers to a page used for planning new sessions.

The third entry leads to a page that lists all the different tracks where the car is tested.

Let's now analyze every single entry mentioned above starting from the session main page and avoiding the last one because it has a simple structure.

Likewise the second layout of the main page described in the previous paragraph(6.4.2), the main layout of the session page, as shown in Figure 6.13a, lists all the sessions planned. Users can search sessions using their ids additionally to an event type.

```
// called whenever the input inside the search bar changes
void filterListByQuery(String query) {
  List<bool> buttonPressed = [
    isAccelerationPressed,
    isSkidpadPressed,
    isEndurancePressed,
    isAutocrossPressed
  ];
```

[3]The menu is created using the hidden drawer menu package.

```
List<String> events = [
  "Acceleration",
  "Skidpad",
  "Endurance",
  "Autocross"
];

// check if an event type is clicked
int indexOfTrue =
  buttonPressed.indexWhere((element) => element);

if (indexOfTrue != -1 && query.isNotEmpty) {
  // event and query
  setState(() {
    _filteredSessionList = sessionList
      .where((element) =>
        element.uid.toString().contains(
              query.toString()
          ) &&
        element.evento == events[indexOfTrue])
      .toList();
    });
} else if (indexOfTrue != -1 && query.isEmpty) {
  // event and !query
  setState(() {
    _filteredSessionList = sessionList
      .where((element) =>
```

```
            element.evento == events[indexOfTrue])
          .toList();
    });
  } else if (indexOfTrue == -1 && query.isNotEmpty) {
    // !event and query
    setState(() {
      _filteredSessionList = sessionList
        .where((element) =>
          element.uid.toString().contains(
            query.toString())
        ).toList();
      });
  } else {
    // !event and !query
    setState(() {
      _filteredSessionList = sessionList;
    });
  }
}
```

In this case `_filteredTeamList` contains all the sessions retrieved from the data source, along with additional information related to event types. Each session is accompanied by a small button that allows users to view the details of the selected session. Clicking this button opens a new page with multiple features and a card displaying all the session information. The layout of this page is described in Figure 6.13c.

The first feature regards the comments related to the session. Managers and foremen can add, modify and delete comments, indicating whether the remarks

were provided by drivers or the team. The layout is described in Figure 6.13d.

The second feature regards the breakages occurred during the session. Managers and foremen can add new breakages, specifying whether the damages were caused by drivers. The layout is described in Figure 6.13e.

The third and fourth features relate to driver participation and the setups used. The layout structure is similar to the first and second feature. However, in this case, when managers and foremen attempt to add a new driver participation or a new used setup, if all the drivers have driven the car or all the setups have been tested, the addition will be blocked. Their layouts are described in Figure 6.13f and 6.13g.

The last feature is the most complex. It allows managers and foremen to modify a session. Since the concept of this feature is similar to the one used for planning a new session, it will be described directly there.

Moving to the add session page, the layout structure is divided in three parts and it is depicted in Figure 6.13h. Users must give accordingly general information in the first part, track information in the second and weather conditions in the third. It is characterized by a widget, called `StepProgressIndicator`, that shows users the step they're currently in. The entire layout has a `bottom navigation bar` with icons that will let users navigate through the different widgets and upload the newest session created with the given inputs to the data source.

Once an user clicks on the upload icon in the third part, if some conditions related to the fillable fields are validated, the `_planSession` method will be executed.

```
void _planSession() async {
  List<bool> buttonPressed = [
    isAccelerationPressed,
```

```
    isSkidpadPressed,
    isEndurancePressed,
    isAutocrossPressed
  ];
  List<String> events = [
    "Acceleration",
    "Skidpad",
    "Endurance",
    "Autocross"
  ];

  await _sessionService.addSession(
    events[buttonPressed.indexWhere(
      (element) => element)
    ],
    newDate,
    newStartingTime,
    newEndingTime,
    newTrack,
    _controllerWeather.text,
    _controllerPressure.text,
    _controllerAirTemperature.text,
    _controllerTrackTemperature.text,
    _controllerTrackCondition.text
  );

  showToast("Sessione pianificata con successo");
```

```
  resetWidget();
}
```

`resetWidget()` is a method that will bring the layout to an initial condition.



(a) Session (b) Drawer (c) Details (d) Comments

(e) Breakages (f) Participation (g) Used setups (h) Plan

Figura 6.13: Session pages

6.4.4 Setup page

By clicking on the bottom-right button within the homepage, users will be redirected to a new page where they can manage setups.

Likewise the session page, the app bar contains an icon, that, when clicked opens a menu with two different entries. The menu is depicted in Figure 6.14b.

The first entry redirects users to the main page of the setup layout described in Figure 6.14a, where all the setups saved are listed. Each element of the list is accompanied by a small button that allows users to view the details of the setup selected.

Clicking this button, managers and foremen are redirected to the page depicted in Figure 6.14c. In this page they can decide whether to modify the selected setup or download inside the 'Download' folder of the device an Excel file that contains the setup parameters. Since the modification layout is similar to the layout used to create a new session, the entire concept behind the business logic will be described afterwards.

The second entry redirects managers and foremen to a page used for creating new setups. The layout used, described in Figure 6.14d, is structured in five different parts. Users must give accordingly wheel parameters in the first part, balance parameters in the second, spring parameters in the third, damper parameters in the fourth and general information in the fifth. It is characterized by a widget called
`StepProgressIndicator`, that shows users the step they're currently in.

The logic behind this page is quite complex. First of all users can decide whether to use existing parameters for all the components or to create new sets of parameters. The procedure for interchanging between existing and non existing parameters is to toggle the `CheckBox` widget. Indeed, every time users clicks on

a `CheckBox`, a method called `_changeStateFrontCheckbox`, where 'Front' is changeable into 'Rear' or 'FrontRight', 'FrontLeft', 'RearRight' and 'RearLeft' for wheels parameters, will be executed.

```
_changeStateFrontCheckbox(bool? newValue) {
  setState(() {
    _useExistingParamsFront = newValue!;
    if (_useExistingParamsFront) {
    // existing parameters
      frontBalance = frontBalanceParams.first;

      _controllerFrontPeso.text =
            frontBalance.peso.toString();
      _controllerFrontFrenata.text =
            frontBalance.frenata.toString();

      balanceProvider.front = frontBalance;
      balanceProvider.existingFront = true;
    } else {
    // non existing parameters
      balanceProvider.front = null;

      _controllerFrontPeso.clear();
      _controllerFrontFrenata.clear();
      balanceProvider.existingFront = false;
    }
  });
}
```

If users want to use existing parameters in a position in which the car component can be present, a `DropdownButton` widget will appear for the position chosen. All of the `DropdownButton` items displayed with their ids, refer to an existing set of parameters for that component in the selected position (e.g., item with 'Id :1' will refer to the component object with the id attribute set to 1).

When users change the `DropdownButton`'s item, a method named similarly to `_changeDropdownItemFront` will be executed.

```
_changeDropdownItemFront(int? balanceId) {
  Balance frontNewBalance =
        frontBalanceParams.where(
          (element) => element.id == balanceId!
        ).first;

  setState(() {
    frontBalance = frontNewBalance;

    balanceProvider.front = frontBalance;
    balanceProvider.existingFront = true;

    _controllerFrontPeso.text =
          frontNewBalance.peso.toString();
    _controllerFrontFrenata.text =
          frontNewBalance.frenata.toString();
  });
}
```

In this example, `frontBalanceParams` is a `List<Balance>` containing all the objects of the `Balance` class in which the attribute 'position' is set to 'Ant'.

Once the method is completely executed, the `TextField` will display the chosen parameters.

Before users navigate through the pages using the `BottomNavigationBar` icons, there is always a check of the parameters by a method called similarly to `_tryPreviousStepFromDamperPage` or `onDamperFromPage`, which will be executed when users attempt to move forward or backward from the page.

```
void _tryPreviousStepFromDamperPage(
List<Damper?> dampers) {
  if (dampers[0] != null) {
    if (dampers[1] != null) {
      setState(() {
        _progress--;
      });
    } else {
      showToast(
        "Specificare gli ammortizzatori posteriori"
      );
    }
  } else {
    showToast(
      "Specificare gli ammortizzatori anteriori"
    );
  }
}
```

These methods check if the parameters that are stored in a `Provider` designated for each component of the car are null or not. If the parameters are null, the

user hasn't filled all the required parameters, and the step will remain the same. Otherwise, the navigation will be allowed for the user.

To set the parameters into the assigned `Provider`, a method called `_checkNewValuesUsedFront` is executed.

```
_checkNewValuesUsedFront(String? text) {
  bool allInputFieldsFilled = true;

  List<String> controllersTexts = [
    _controllerFrontPeso.text,
    _controllerFrontFrenata.text
  ];

  final Iterator<String> iterator =
        controllersTexts.iterator;

  while (iterator.moveNext()) {
    if (iterator.current.isEmpty) {
      allInputFieldsFilled = false;
    }
  }
  if (allInputFieldsFilled) {
    if (
      double.tryParse(
        _controllerFrontFrenata.text
      ) != null
    ) {
      if (
```

```
        double.tryParse(
          _controllerFrontPeso.text
        ) != null
      ) {
        var result = findFrontBalanceFromExistingParams(
           _controllerFrontPeso.text,
           _controllerFrontFrenata.text
        );

        Balance balance;

        if (result != null) {
          balance = result;
        } else {
          balance = Balance(
              id: 0,
              posizione: "Ant",
              peso: double.parse(
                _controllerFrontPeso.text
              ),
              frenata: double.parse(
                _controllerFrontFrenata.text
              )
          );
        }
        balanceProvider.front = balance;
        balanceProvider.existingFront = false;
```

```
        } else {
          showToast(
           "Il peso anteriore deve rappresentare un numero"
          );
        }
      } else {
        showToast(
         "La frenata anteriore deve rappresentare un numero"
        );
      }
    }
}
```

This method checks if all the text fields associated with a component in a specific position are filled. If they're not filled, the `Provider` will have a `null` object instead of a real instance of the object needed. If they're all filled instead, if there's an existing object with the given parameters, it will set the corresponding field within the `Provider`, otherwise a new object will be created with an id initially set to '0' and then assigned to the `Provider`.

Once all the parameters are defined, if users click on the upload icon, a method called `_createSetup` will be executed. This time, since the method is pretty long, the snapshot will show the business logic for only one component of the car.

```
Future<void> _createSetup() async {
    List<Wheel> wheels = [
      frontRightWheel,
      frontLeftWheel,
      rearRightWheel,
```

```
  rearLeftWheel
];
List<int> wheelIds = [0, 0, 0, 0];
.
.

final WheelService _wheelService = WheelService();
.
.

await Future.forEach(wheels.asMap().entries,
    (MapEntry<int, Wheel> entry) async {
  final index = entry.key;
  final wheel = entry.value;

  if (wheel.id == 0) {
    wheelIds[index] =
          await _wheelService.addNewWheel(wheel);
  } else {
    wheelIds[index] = wheel.id;
  }
});
.
.

final wheelsUsed = <Wheel>[];
wheels.asMap().forEach((index, wheel) {
```

```
  wheelsUsed.add(Wheel(
      id: wheelIds[index],
      codifica: wheel.codifica,
      posizione: wheel.posizione,
      frontale: wheel.frontale,
      superiore: wheel.superiore,
      pressione: wheel.pressione));
});
.
.

await _setupService.createSetup(
  wheelsUsed,
  balanceUsed,
  springsUsed,
  dampersUsed,
  genInfosUsed
);

showToast("Setup creato con successo");

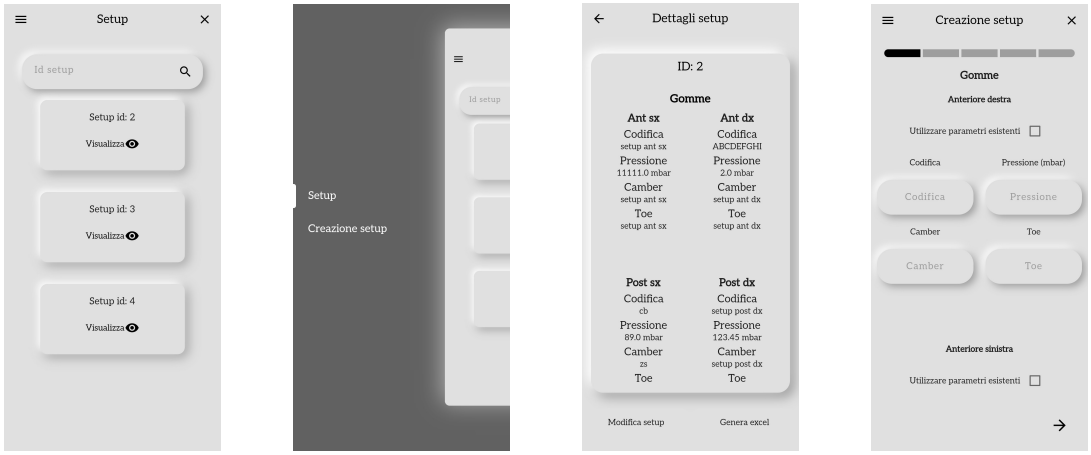
Navigator.popAndPushNamed(context, '/setup',
    arguments: widget.loggedMember);
```

At the beginning, two lists are created for each car component. The first one for the actual objects and the second one for the ids of these objects. Then, for each component, an object of a `service` class is instantiated. Since the objects saved inside the different `Providers` can have an id equal to '0' in case those records

are not saved within Firestore[4], they should be added to the database to create a new setup correctly. So the method iterates over all the elements of each list of objects checking if the element has an id different from '0'. If it's different, then its id is saved within the list of ids of that component in the same position of the corresponding object saved inside the list of objects. Instead, when the id is equal to '0', a new record is saved inside the database, and the `addNewWheel` method will return the id of the stored object once the creation is done. This id will then be positioned inside the list of ids.

Then a new list is created for each car component. This list will contain all the objects saved inside the list of objects, having an id equals to the ones that are instead saved inside the list of ids.

These lists will be passed to the `createSetup` method of the `SetupService`, which will call the same method name of the repository to create the newest setup.



(a) Setup (b) Drawer (c) Details (d) Plan

Figura 6.14: Setup pages

[4]Firebase database

6.4.5 Problem page

By clicking on the top-right button within the homepage, users will be redirected to a new page where they can either manage an existing problem or add a new problem. The layout of the page is depicted in Figure 6.15a.

To add a new problem, they can simply click on the button positioned at the bottom of the page, and inside the dialog that pops up, specify the description. If the description entered matches with another description of an existing problem, the creation will be blocked.

Instead, to manage a problem, users have to click on the button of an element listed. Once the button is clicked a new page will appear. The layout is divided into two different pages and is described in Figure 6.15b.

The first one gathers and display all the setups in which the problem occurs. Here, users can decide to add a new occurring problem or to change the state of an existing one to *solved*.

To add a new occurring problem, users have to click on a button similar to the one used to create a new problem. In this case, though, in the dialog that pops up, in addition to a text field in which they can specify a description, there's also a `DropdownButton` displaying all the setup's ids in which the selected problem was neither found nor solved. If there are no setups left, and so if in all of the saved setups the problem was found or solved, the creation of a new occurring problem will be blocked.

To change the state of the problem users can click on the button contained in each element of the list and then, if needed, fill a new description field.

Instead, the second page described in Figure 6.15c, lists all the setups in which the problem was solved. Likewise in the occurring problem page, even if there's not a button that will add a new solved problem, each element of the list has a

button, that, when clicked, will change the state of the problem from saved to occurring.



(a) Problem (b) Occurring (c) Solved

Figura 6.15: Problem pages

6.4.6 Telemetry page

By clicking on the bottom-left button within the homepage, users will be redirected to a new page depicted in Figure 6.16a where they can view the telemetry.

To view the data collected during the sessions, users initially have to specify the session and the setup of which they want to see the telemetry.

If, after selecting a session from a `DropdownButton` that lists all the ids of the sessions saved, the button used to display the charts disappears, it means that no setups where tested during the session selected, and therefore, no data was collected.

Instead, if users change the session id and the button remains visible, it means that they can choose the setups from another `DropdownButton` that lists all the ids of the setups used during that session. If then users click on the button used to display the charts and the page still stays, it indicates that no data was collected for the setup selected during that session.

In the other hand, when data is collected for that setup during the session selected, a new page having the layout described in Figure 6.16b pops up.

In the app bar of this page there's an icon that, when clicked, opens a menu[5] with multiple different entries. The menu is depicted in Figure 6.16c.

Each of them redirects users to a new page in which a different chart is displayed. All the charts come along with a legend positioned at the bottom of the cart itself.

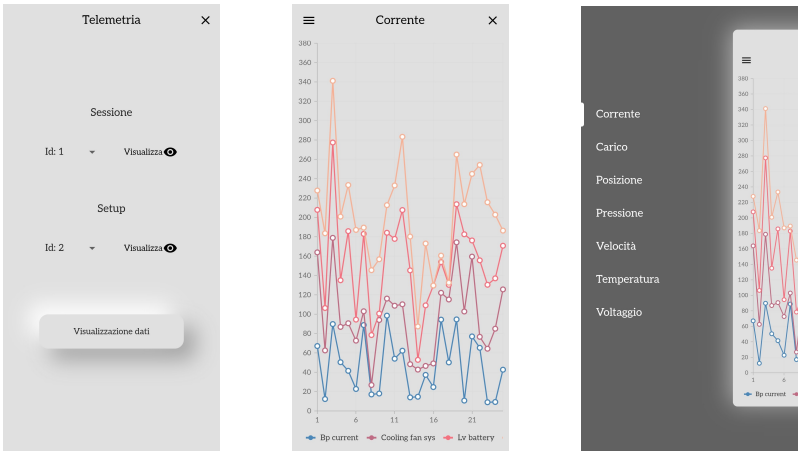
Users can toggle the visibility of the different 'lines' by clicking on the name associated with the line color displayed on the legend of the chart.

The code written to display the data is similar for all the different charts reachable from the menu entries.

```
SfCartesianChart(
  tooltipBehavior: _tooltipBehavior,
  legend:
    Legend(
      isVisible: true,
      position: LegendPosition.bottom
    ),
  series: <ChartSeries>[
    StackedLineSeries<Current, int>(
      dataSource: chartData,
```

[5]The menu is created using the hidden drawer menu package

```
      xValueMapper: (Current curr, _) => curr.id,
      yValueMapper: (Current curr, _) => curr.bpCurrent,
      name: "Bp current",
      markerSettings: MarkerSettings(isVisible: true)
    ),
    .
    .
    .
    ],
    zoomPanBehavior: ZoomPanBehavior(
      enablePinching: true,
      enableDoubleTapZooming: true,
      enablePanning: true,
    ),
);
```



(a) Telemetry (b) Charts (c) Drawer

Figura 6.16: Telemetry pages

6.4.7 New member page

By clicking on the bottom-center button withing the homepage, managers will be redirected to a new page where they can add new members.

The layout is formed by two pages and is described in Figure 6.17.

In the first one users can define personal details of the newest member, such as student ID number, first name and last name.

In the second one instead it can only be defined the role that will be covered and, in case, the workshop area.

Once users click on the icon to save the member, if all the information are filled correctly, the credentials will be sent to the email correlated to the student ID number entered.

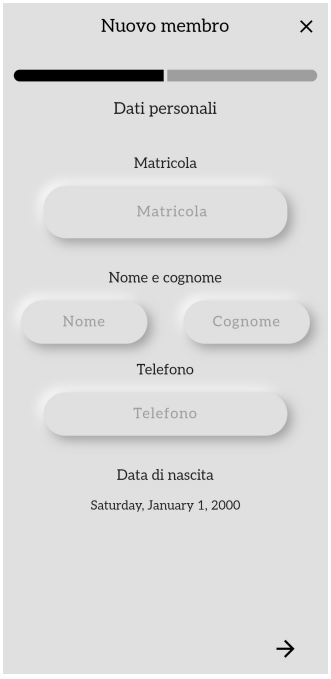


Figura 6.17: New member page

Capitolo 7

Database and API using Spring Boot

The application was designed with future development and maintenance in mind. Consequently, while Firebase served as an excellent tool for laying the foundation of the application, it may not be a suitable solution for the back-end portion in the future, as there will be an urgent need to store and retrieve vast amounts of data within a database. The best solution would be to utilize a software capable of executing queries more efficiently than Firebase.

For this reason, a future development will require a different tool from **Firebase**.

Here comes **Spring Boot** [8], a framework utilized for building Java-based applications. It is built on top of the **Spring** framework, which enables users to create **REST APIs** with minimal configuration. A REST API, also called 'RESTful API', is an architectural style for an API that uses **HTTP** requests to access and use data.

To get started with this framework, it is recommended to check the guidelines within the document [8].

7.1 ER diagram model of the database

An **ER diagram**, also know as Entity-Relationship diagram, is used to create a conceptional model of a database. As the name recall, this type of diagram is made up by two main features surrounded by other characteristics.

The first component is defined by the concept of **Entity**. Entities represent distinguishable objects that can exist independently. For example, in real life, a tree can be considered an entity, while a leaf cannot. Within the diagram, entities are depicted as rectangles.

The second component is defined by the concept of **Relationship**. Relationships represent associations between two or more entities. Unlike entities, relationships cannot exist independently; they only exist within the defined connections between those entities. Within the diagram, relationships are depicted as diamonds.

The database used for this `Spring Boot` application is structured based on the ER diagram depicted in Figure 7.1.

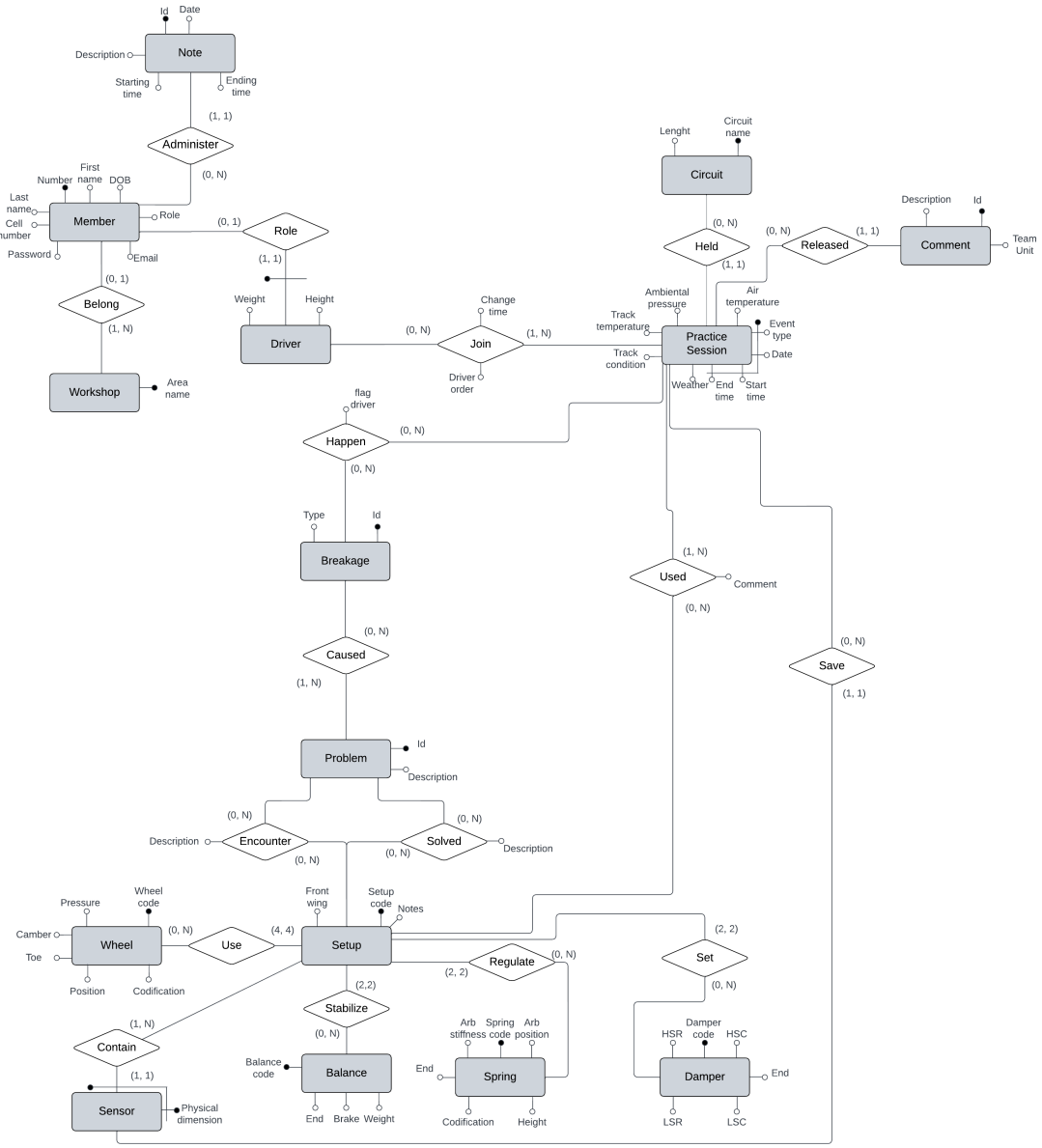


Figura 7.1: Spring Boot - ER diagram

Entities

Member

- *Description* → Represents individuals involved in the team
- *Attributes* → Student ID (Primary key, Int), First name (String), Last name (String), Date of birth (Date), Email (String), Phone number (Int), Password (String), Role (String), Role (Foreign key, String)

Workshop

- *Description* → Represents all the workshop ares of the team
- *Attributes* → Area name (Primary key, String)

Note

- *Description* → Represents notes created by members
- *Attributes* → Id (Primary key, Int), Date (Date), Starting time (Time), Ending time (Time), Description (String), Member (Foreign key, Int)

Driver

- *Description* → Represents members who are drivers
- *Attributes* → Id (Primary key, Int), Member (Foreign key, Int), Weight (Decimal), Height (Decimal)

Practice session

- *Description* → Represents practice sessions in which the car is tested

- *Attributes* → Id (Primary key, Int), Event (String), Date (Date), Start time (Time), End time (Time), Weather (String), Ambient pressure (Decimal), Air temperature (Decimal), Track temperature (Decimal), Track condition (String), Track (Foreign key, String)

Circuit

- *Description* → Represents racetracks
- *Attributes* → Name (Primary key, String), Length (Decimal)

Comment

- *Description* → Represents comments made during practice sessions
- *Attributes* → Id (Primary key, Int), Description (String), Flag (String), Session (Foreign key, Int)

Breakage and Problem

- *Description* → Represents types of breakages and problems
- *Attributes* → Id (Primary key, Int), Description (String)

Wheel

- *Description* → Represents sets of wheels used in vehicle setups
- *Attributes* → Id (Primary key, Int), Codification (String), Position (String), Camber (String), Toe (String), Pressure (Decimal)

Balance

- *Description* → Represents sets of balance components used in vehicle setups
- *Attributes* → Id (Primary key, Int), Position (String), Weight (Decimal), Brake (Decimal)

Spring

- *Description* → Represents sets of springs used in vehicle setups
- *Attributes* → Id (Primary key, Int), Position (String), Codification (String), Height (Decimal), Arb position (String), Arb stiffness (String)

Damper

- *Description* → Represents sets of dampers used in vehicle setups
- *Attributes* → Id (Primary key, Int), Position (String), Hsr (Decimal), Lsr (Decimal), Hsc (Decimal), Lsc (Decimal)

Sensor

- *Description* → Represents sensors used in data collection
- *Attributes* → Id (Primary key, Int), Setup (Foreign key, Int), Session (Foreign key, Int), Sensor data (Decimal)

Setup

- *Description* → Represents vehicle setups

- *Attributes* → Id (Primary key, Int), Front wing (String), Notes (String), Front right wheel (Foreign key, Int), Front left wheel (Foreign key, Int), Rear right wheel (Foreign key, Int), Rear left wheel (Foreign key, Int), Front balance (Foreign key, Int), Rear balance (Foreign key, Int), Front spring (Foreign key, Int), Rear spring (Foreign key, Int), Front damper (Foreign key, Int), Rear damper (Foreign key, Int)

Relationships

Belong

- *Description* → Describes the relationship between workshops and members, indicating that each workshop can have multiple members, and each member belongs to at most one workshop

- *Cardinality* → Workshop (One to Many) - Member (One to One)

Administer

- *Description* → Indicates that a member can administer multiple notes, while each note is administered by a single member

- *Cardinality* → Note (One to One) - Member (One to Many)

Role

- *Description* → Represents the association between members and drivers, showing that each member can be a driver, and each driver is a member

- *Cardinality* → Driver (One to One) - Member (One to One)

Join

- *Description* → Defines the relationship between drivers and practice sessions, signifying that drivers can participate in multiple practice sessions, and each session can have multiple participating drivers
- *Cardinality* → Driver (One to Many) - Practice session (One to Many)
- *Attributes* → Driver order (Int), Change time (Time)

Held

- *Description* → Depicts that circuits can host multiple practice sessions, with each session being held at one circuit
- *Cardinality* → Circuit (One to Many) - Practice session (One to One)

Released

- *Description* → Shows the relationship between comments and practice sessions, indicating that each comment is released for a single practice session
- *Cardinality* → Comment (One to One) - Practice session (One to Many)

Happen

- *Description* → Represents the occurrence of various breakagesduring practice sessions, illustrating that multiple types of breakages can happen in each session
- *Cardinality* → Breakage (One to Many) - Practice session (One to Many)
- *Attributes* → Flag driver (Bool)

Caused

- *Description* → Describes the connection between types of breakages and the problems they can cause, showing that multiple types of breakages can cause multiple problems, and vice versa
- *Cardinality* → Breakage (One to Many) - Problem (One to Many)

Encounter

- *Description* → Indicates that problems can be encountered by multiple setups, and each setup can encounter multiple problems
- *Cardinality* → Problem (One to Many) - Setup (One to Many)
- *Attributes* → Description (String)

Solved

- *Description* → Shows that problems can be solved in multiple setups, and each setup can solve multiple problems
- *Cardinality* → Problem (One to Many) - Setup (One to Many)
- *Attributes* → Description (String)

Use

- *Description* → Represents the utilization of sets of wheels in vehicle setups, with each setup using multiple sets of wheels, and each set of wheels being used in multiple setups
- *Cardinality* → Wheel (One to Many) - Setup (Four to Four)

Stabilize

- *Description* → Depicts the utilization of sets of balance components in vehicle setups, where each setup uses multiple sets of balance components, and each set of balance components is used in multiple setups
- *Cardinality* → Balance (One to Many) - Setup (Two to Two)

Regulate

- *Description* → Shows the utilization of sets of spring components in vehicle setups, indicating that each setup uses multiple sets of springs, and each set of springs is used in multiple setups
- *Cardinality* → Spring (One to Many) - Setup (Two to Two)

Set

- *Description* → Represents the usage of sets of damper components in vehicle setups, illustrating that each setup uses multiple sets of damper components, and each set of dampers is used in multiple setups
- *Cardinality* → Damper (One to Many) - Setup (Two to Two)

Contain

- *Description* → Depicts the containment of sensor types within setups, with each setup containing at least one type of sensor, and each sensor type being related to one setup
- *Cardinality* → Setup (One to Many) - Sensor (One to One)

Used

- *Description* → Illustrates that setups can be used in multiple practice sessions, and each session can involve multiple setups
- *Cardinality* → Setup (One to Many) - Practice session (One to Many)
- *Attributes* → Comment (String)

Save

- *Description* → Represents the storage of sensor data for each practice session, where data gathered from sensors are related to one session, and vast amounts of data can be stored for each session
- *Cardinality* → Sensor (One to One) - Practice session (One to Many)

7.2 Architectural design pattern

The pattern chosen for this application is the exact same pattern utilized for the `Flutter` project, so the **service repository pattern** in combination with a simple `Model-View-Controller`.

The MVC architecture promotes a separation of concerns by splitting your application into these distinct components. Models represent the data and business logic. Services contain the application's core logic. Controllers handle the user interface and interaction. Repositories manage data access and persistence. This separation helps make the application more maintainable, testable, and scalable, as changes in one component typically have minimal impact on the others.

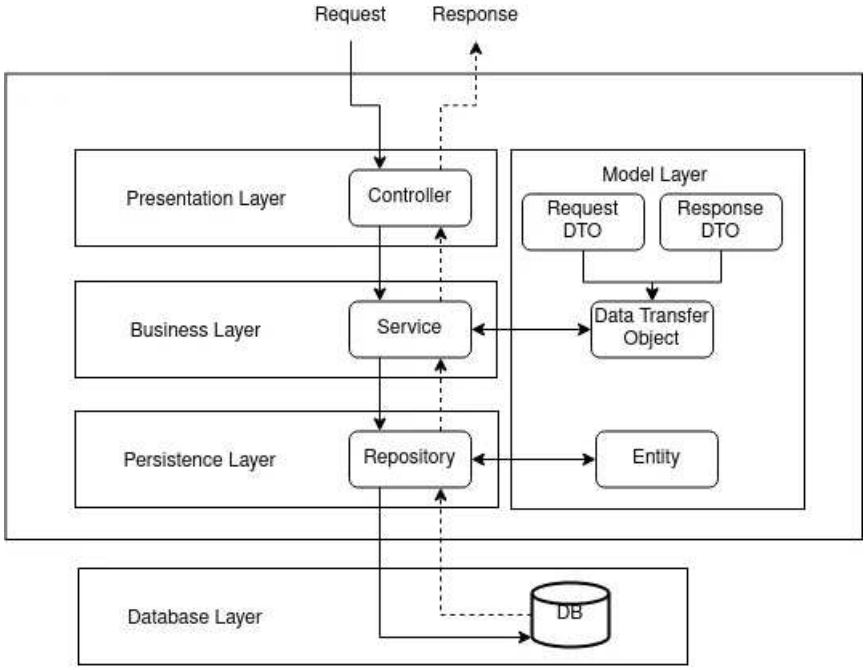


Figura 7.2: Spring Boot architecture - Project folder

Let's break down the structure of the different components of the entire project with some examples.

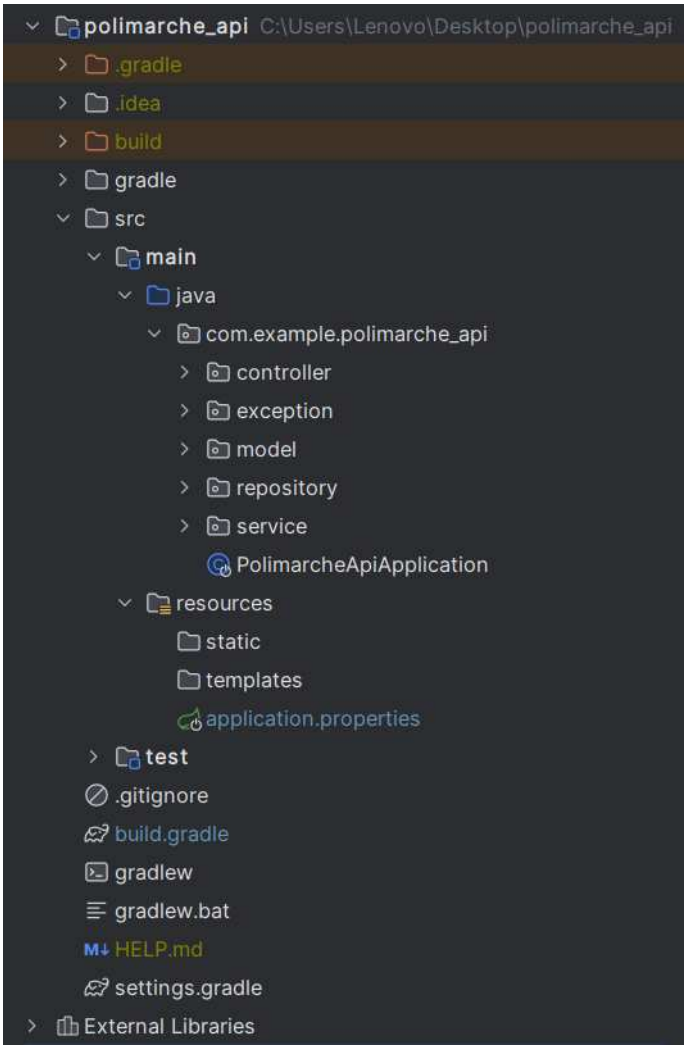


Figura 7.3: Spring Boot architecture - Project folder

7.2.1 Model

Every model class is defined by the `@Entity` annotation.
`@Table(name = 'value')` annotation can be used to define the name of the table inside the database that the entity should represent. The name of the relation takes the place of `'value'` inside the annotation.

When using `@Entity` annotation it's recommended to add the `@Id` annotation. In this way, since spring boot utilizes `JPA repositories` and `Hibernate` for database persistence, the primary key of the corresponding database table can be defined.

If the `Id` is auto incremental, `@GeneratedValue` annotation must be used. One of its properties define the strategy for the creation of records inside the database. Indeed, `strategy = GenerationType.IDENTITY` specifies that every time a new instance is saved into the database, its ID will be automatically generated.

Every model class should also define all the `getters` and `setters` since all the attributes are private, and override `hashCode()`, `equals()` and `toString()` methods.

```
@Entity
@Table(name = "bilanciamento")
public class Balance {

  @Id
  @GeneratedValue(
    strategy = GenerationType.IDENTITY
  )
  private Integer id;
```

```
  private String posizione;
  private Double frenata;
  private Double peso;

  public Balance(
    Integer id,
    String posizione,
    Double frenata,
    Double peso
  ) {
    this.id = id;
    this.posizione = posizione;
    this.frenata = frenata;
    this.peso = peso;
  }

  public Balance() {}
}
```

7.2.2 Repository

Every repository class is defined by the `@Repository` annotation.

All the repositories are interfaces extending a class called `JpaRepository<Object, Id_type>`.

They are interfaces because all the methods declared define different database operations, like queries, updates, deletions etc. The implementation is automatically generated by Spring Data JPA at runtime based on the method names and signatures(e.g., 'findObjectByAttribute(Type value)' will find all the records of the database where the attribute is equal to the value given).

Additionally they extend `JpaRepository`. It is a generic interface that defines common database operations for managing entities. The first generic parameter('Object') specifies the entity type that the repository manages, and the second generic parameter('Id_type') specifies the type of the entity's primary key.

```
@Repository
public interface BalanceRepository
    extends JpaRepository<Balance, Integer> {

    List<Balance> findBalanceByPosizione(
      String position
    );
}
```

7.2.3 Service

Each service class is defined by `@Service` annotation.

Since a service is created for a single model, to define data operations like creations or updates, every single service has always an attribute referring to the corresponding repository of that model class.

```
@Service
public class BalanceService {

  private final BalanceRepository balanceRepository;

  public BalanceService(
    BalanceRepository balanceRepository
  ) {
    this.balanceRepository = balanceRepository;
  }

  .
  .

}
```

7.2.4 Controller

Each controller is defined by two annotations. The first one is `@RestController` and the second one is `@RequestMapping('path')`.

They declare a `RESTful controller` and the base path for all the request mappings within this controller.

A controller for model class 'Object' will have at least an attribute defining a reference to the service of the same 'Object' class.

All the methods defined within a controller must have an annotation declaring the type of action users want to fulfill. There are different annotations such as `GetMapping`, `PostMapping`, `PutMapping` and `DeleteMapping`. They are used respectively for getting data, creating records into the database, updating existing records, and deleting records. Each one can also define a new path from which the request sent will be handled by the controller.

```
@RestController
@RequestMapping("api/v1/balance")
public class BalanceController {

  private final BalanceService balanceService;

  public BalanceController(
    BalanceService wheelService
  ) {
    this.balanceService = wheelService;
  }
```

```
  @GetMapping
  public List<Balance> getAllBalances(){
    return balanceService.getAllBalances();
  }

  @GetMapping("/position/{position}")
  public List<Balance> getAllBalancesByPosition(
    @PathVariable String position
  ){
    return balanceService.getAllBalancesByPosition(
      position
    );
  }
}
```

7.3 API development

The development of the API followed the principles described in the previous section(7.2). The correct execution of the different calls made through the `HTTP` protocol was tested using `JUnite5`, which will be explained in the next section, and `Postman` (check section 7.4.2).

To understand if the development was done accurately step by step, immediately after the implementation of each model, its repository, service and controller were also coded. In this way, whenever some bugs appeared, they could have been solved immediately.

On this basis, the development process can be considered circular. Without analyzing the entire project, let's break down a single cycle.

The case study will be represented by 'Join' relation of the ER diagram.

7.3.1 Database connection

Initially, it is required to connect the application to a MySQL database, and Spring Boot simplifies this process drastically.

Inside the `application properties` file, some properties of the database should be specified .

```
server.port=YOUR_PORT


spring.datasource.url =
    jdbc:mysql://localhost:3306/YOUR_DATABASE_NAME
spring.datasource.username =
    YOUR_USERNAME
spring.datasource.password =
    YOUR_PASSWORD
// driver to connect to the database
spring.datasource.driver-class-name =
    com.mysql.cj.jdbc.Driver


spring.jpa.properties.hibernate.dialect =
    org.hibernate.dialect.MySQL8Dialect
```

The URL `jdbc:mysql://localhost:3306/YOUR_DATABASE_NAME` redirects to the port of the machine in which the MySQL database, specified by 'YOUR_DATABASE_NAME', runs.

7.3.2 Model development

Since the model should resemble the corresponding entity described by the figure 7.1, all the attributes defined within the class will refer to the attributes provided

from the ER diagram.

Within the application, 'Join' relation was refactored to 'Participation' for reasons of comprehension and concept clarification.

```
@Entity
@Table(name = "partecipazione")
public class Participation {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @ManyToOne
  @JoinColumn(name = "pilota")
  private Driver pilota;

  @ManyToOne
  @JoinColumn(name = "sessione")
  private PracticeSession sessione;

  private Integer ordine;

  private String cambio_pilota;
}
```

This snapshot provides the structure of 'Participation' model class.

Two further annotations, different from the ones described in the subsection 7.2.1, were utilized.

The first one is `@ManyToOne`. It's a `Jpa annotation` used to map a many-to-one relation between entities and/or relations. In simple terms, it means that inside a database, multiple rows of a relation are mapped to one row of another table. In this case, multiple `Objects` of 'Participation' can have the same reference to 'Driver' or 'PracticeSession'.

The second one is `@JoinColumn`. It's used to mark a column as a join column for an entity association or an element collection. `name` property specifies the name of the `foreign key` column of 'Participation'.

7.3.3 Repository development

`Repositories` have already been described carefully in the subsection 7.2.2.

Here is only provided a snapshot with the definition of the repository class and some of its methods.

```
@Repository
public interface ParticipationRepository
      extends JpaRepository<Participation, Integer> {

  Integer countParticipationBySessioneId(
    Integer sessioneId
  );

  List<Participation> findBySessioneId(Integer sessione);
  Boolean existsBySessioneId(Integer sessione);
}
```

7.3.4 Service development

As for the repository, since `services` have already been described accurately in the subsection 7.2.3, here is provided a snapshot with the definition of the service class and the explanation of some of its methods.

```
@Service
public class ParticipationService {

  private final
    ParticipationRepository participationRepository;

  private final DriverRepository driverRepository;

  private final
    ParticipationDTOMapper participationDTOMapper =
      new ParticipationDTOMapper();

  public ParticipationService(
    ParticipationRepository participationRepository,
    DriverRepository driverRepository
  ) {
    this.participationRepository =
          participationRepository;
    this.driverRepository =
          driverRepository;
  }
```

```
  public List<ParticipationDTO> getAllParticipation() {
    return participationRepository.findAll()
      .stream()
      .map(participationDTOMapper)
      .collect(Collectors.toList());
  }
}
```

It can be noticed that, other than the expected repositories references as attributes, there's a new attribute of type `ParticipationDTOMapper`.

As the name of the type may suggest, `ParticipationDTOMapper` is a `DTO mapper`.

Generally, a `DTO mapper` is a component or class in a software application that is responsible for mapping data between objects of different types, typically between model objects, usually called domain model objects, and `DTOs`, where a `DTO` is a class that contains a subset of the data from a model object.

They are often used as the objects that are passed as the result of an HTTP request. In this way, unnecessary or confidential data won't be sent to the `client` who sent the request.

Indeed, `getAllParticipation` method, which is called from the controller returns a `List<ParticipationDTO>` and not a `List<Participation>`.

Within the Spring Boot application, all of the `DTOs mapper` are marked by the `@Service` annotation and implements an interface.

This interface is `Function<ModelObject, ModelObjectDto>`. It represents a function that takes one argument and produces a result. In our case takes a model object and transforms it into a DTO object.

The transformation is done by `apply` method. Every class that implements a `Function` must override the method.

```
@Service
public class ParticipationDTOMapper implements
    Function<Participation, ParticipationDTO> {
  @Override
  public ParticipationDTO apply(
    Participation participation
  ) {
    Driver driver = participation.getPilota();
    MemberDTO memberDTO = new MemberDTO(
      driver.getMembro().getMatricola(),
      driver.getMembro().getNome(),
      driver.getMembro().getCognome(),
      driver.getMembro().getData_di_nascita(),
      driver.getMembro().getEmail(),
      driver.getMembro().getNumero_telefono(),
      driver.getMembro().getRuolo(),
      driver.getMembro().getReparto()
    );
    return new ParticipationDTO(
      participation.getId(),
      new DriverDTO(
        memberDTO,
        driver.getPeso(),
        driver.getAltezza()
      ),
      participation.getSessione(),
      participation.getOrdine(),
```

```
      participation.getCambio_pilota()
    );
  }
}
```

7.3.5 Controller development

As for the repository and service, since `controllers` have already been described accurately in the subsection 7.2.4, here is provided a snapshot with the definition of the controller class and the explanation of some of its methods.

```
@RestController
@RequestMapping("/api/v1/participation")
public class ParticipationController {

  private final ParticipationService participationService;

  public ParticipationController(
    ParticipationService participationService
  ) {
    this.participationService = participationService;
  }

  @GetMapping
  public List<ParticipationDTO> getAllParticipation(){
    return participationService.getAllParticipation();
  }

  @GetMapping("/session/{session}")
```

```
  public List<ParticipationDTO>
    getParticipationBySession(
      @PathVariable Integer session
  ) {
    return participationService.getParticipationBySession(
      session
    );
  }

  @PostMapping
  public ResponseEntity<Integer> addNewParticipation(
    @RequestBody NewDriverParticipation request
  ) {
    Integer id =
          participationService.addNewParticipation(
            request
          );
    return new ResponseEntity<>(id, HttpStatus.CREATED);
  }
}
```

All the methods created define a combination of the type of the `Http` request and the path for the mapping of the request sent.

The first method (`getAllParticipation`) returns a list of all the `ParticipationDTO` got from the homonymous method of the `service` class whenever the request is sent to path `/api/v1/participation`.

The second method (`getParticipationBySession`) returns a list of all the `ParticipationDTO` in which the session id is the same as the one ente-

red as a parameter within the URL of the request. {session} indicates that a parameter is allowed at the end of the path, and in case a value is passed, it will be stored inside a variable called session. The path of the request is /api/v1/participation/session/{session}.

The third and last method shown (addNewParticipation) is responsible for the creation of a new record within the database. Differently from any GET request, a POST request sends data to the server through its body. Within the body can only be sent data which structure represents a record called NewDriverParticipation. A record is a class-like construct designed to simplify the creation of classes whose main purpose is to store data.

```
public record NewDriverParticipation(
  @JsonProperty("pilota") Driver pilota,
  @JsonProperty("sessione") PracticeSession sessione,
  @JsonProperty("ordine") Integer ordine,
  @JsonProperty("cambio_pilota") String cambio_pilota
) {}
```

@JsonProperty annotation specifies how the fields of the record should be serialized to and serialized from JSON.

7.3.6 API calls

Below is a list of API calls implemented for each model class. While all models share a common pattern, consisting of calls that retrieve all records from the database, create new records, modify, and potentially delete existing ones.

Additionally, each model has its set of specific calls to achieve specific functions. Those are well defined and analyzed in the table.

API call	Http method	Parameters	Description
/api/v1/track			**Track**
/api/v1/session			**Session**
/{id}	GET	'id' is `Integer`	Return the session with the id given
/by-event/{event}	GET	'event' is `String`	Return all sessions for the event given
/api/v1/workshop			**Workshop**
/api/v1/member			**Member**
/login	POST	'matricola' and 'password' within the request's body	Return the member with the credentials sent or throw an `Exception`
/api/v1/note			**Note**
/member /{matricola}	GET	'matricola' is Integer	Return all notes of the member defined
/api/v1/comment			**Comment**
/session/{id}	GET	'id' is `Integer`	Return all comments released during the session given

/api/v1/driver			Driver
/{matricola}	GET	'matricola' is `Integer`	Return the driver with the 'matricola' given or throw an `Exception`
/api/v1/participation			**Participation**
/session /{session}	GET	'session' is `Integer`	Return all participation during the session given
/driver /{matricola}	GET	'matricola' is `Integer`	Return all participation of the driver specified
/driver-session	GET	'matricola' and 'sessionId' should be specified as `Params`	Return all participation of the driver specified during the session given
/api/v1/breakage			**Breakage**
/api/v1/breakage-happened			**Breakage happened**
/driver-fault	GET	-	Retrieve all breakages happened caused by drivers
/session/{id}	GET	'id' is `Integer`	Return all breakages happened during the session given

/breakage/{id}	GET	'id' is `Integer`	Return all breakages happened with the defined breakage
/api/v1/problem			**Problem**
/api/v1/problem-encountered			**Occurring problem**
/setup/{id}	GET	'id' is `Integer`	Return all problems encountered by the setup given
/problem/{id}	GET	'id' is `Integer`	Return all problems encountered of the defined problem
/api/v1/problem-solved			**Solved problem**
/setup/{id}	GET	'id' is `Integer`	Return all problems solved into the setup given
/problem/{id}	GET	'id' is `Integer`	Return all problems solved of the defined problem
/api/v1/wheel			**Wheel**
/position /{position}	GET	'position' is `String`	Return all wheels with the defined position
/api/v1/balance			**Balance**
/position /{position}	GET	'position' is `String`	Return all balance parameters with the defined position
/api/v1/spring			**Spring**

/position /{position}	GET	'position' is `String`	Return all springs with the defined position
/api/v1/damper			**Damper**
/position /{position}	GET	'position' is `String`	Return all dampers with the defined position
/api/v1/setup			**Setup**
/{id}	GET	'id' is `Integer`	Return the given setup
/api/v1/used-setup			**Used setup**
/setup/{id}	GET	'id' is `Integer`	Return all sessions in which the setup specified was used
/session/{id}	GET	'id' is `Integer`	Return all setups used during the session specified
/api/v1/current			**Current**
/api/v1/load			**Load**
/api/v1/position			**Position**
/api/v1/pressure			**Pressure**
/api/v1/speed			**Speed**
/api/v1/temperature			**Temperature**
/api/v1/voltage			**Voltage**

Tabella 7.1: API calls

7.4 API test

As mentioned in the previous section (7.3), the API was tested through the use of Postman and JUnit5.

7.4.1 JUnit5

JUnit is an open framework used to test java applications. It allows developers to write and execute automated tests. Check the article *'JUnit Framework For Unit Testing [9]'* for further information.

The part of the application tested regards a subset of services. This choice was taken because if repositories and their methods are built automatically by JPA, controllers don't contain any business logic. Furthermore, controllers are also tested using Postman.

However, testing a service requires some attentions. Indeed, if a test is executed over a bare service, whenever one of its methods that needs to operate with the database is called, data produced by the test will not be related to the application, and, if it will be stored within the db, it can compromise the efficiency of the entire software and the execution speed of the test.

For this reason tests are often done in combination of H2, an open source SQL database used as in-memory database. The difference between in-memory databases and normal databases is that if in normal database all data is stored on disk and is always persisted to disk, in in-memory databases all data is stored in main memory and is volatile. It comes clear that, since CPU is less used in in-memory databases, the execution time of tasks is lower compared to normal databases.

H2 requires some configuration settings within the application.properties file inside resources sub folder of test folder.

```
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

Let's analyze and describe a part of the test of `ParticipationService`.

```
@ExtendWith(MockitoExtension.class)
class ParticipationServiceTest {

  @Mock
  private ParticipationRepository participationRepository;

  @Mock
  private DriverRepository driverRepository;

  private ParticipationService underTest;

  @BeforeEach
  void setUp() {
    this.underTest = new ParticipationService(
      participationRepository,
      driverRepository
    );
  }
```

```
  @Test
  void canGetAllParticipation() {
    //given

    // when
    underTest.getAllParticipation();

    // then
    verify(participationRepository).findAll();
  }
}
```

`@ExtendWith(MockitoExtension.class)` annotation is used to extend the test class and utilizes `Mockito`. It allows multiple features, such as the creation of mock objects, the definition of mock behaviors, and the verification of interactions with those mock objects in your unit tests. Each mock object is marked by `@Mock` annotation.

In all of the tests, `underTest` variable is the `service` which methods have to be tested.

`@BeforeEach` is used to signal the code that should be executed before each test method of the specific class.

`@Test` actually defines a test. All the test are divided in three steps.

The first step is `given`. Here objects are created that are required as return values for mocked method calls or as input parameters to the method under test. Furthermore, the mocked method calls as such are prepared in this section.

The second step is `when`. In this section basically the tested method are called.

The third and last step is `then`. Here assertions on any results from the execution step are implemented. In addition, it can be checked if certain (mocked)

method calls have happened through `verify` method.

7.4.2 Postman

`Postman` is a tool that allows developers to easily create, test, share, and document APIs.

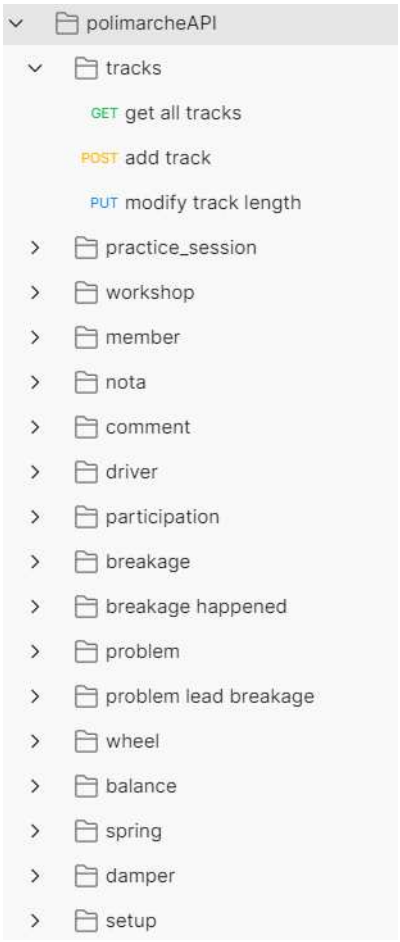


Figura 7.4: Test API - Postman

As it can be seen, one of the features of Postman is to store requests into files which can then be organized in folders.

To verify if a request returns the proper output, the data received can be compared to data stored within the database. If data matches, the implementation of the method of the controller works properly.

	id	pilota	ordine	cambio_pilota	sessione
1	21	5	2	<null>	44
2	23	1	3	<null>	44
3	24	1	1	<null>	45
4	25	8	1	00:00:21.932	44

Figura 7.5: Test API - Database data

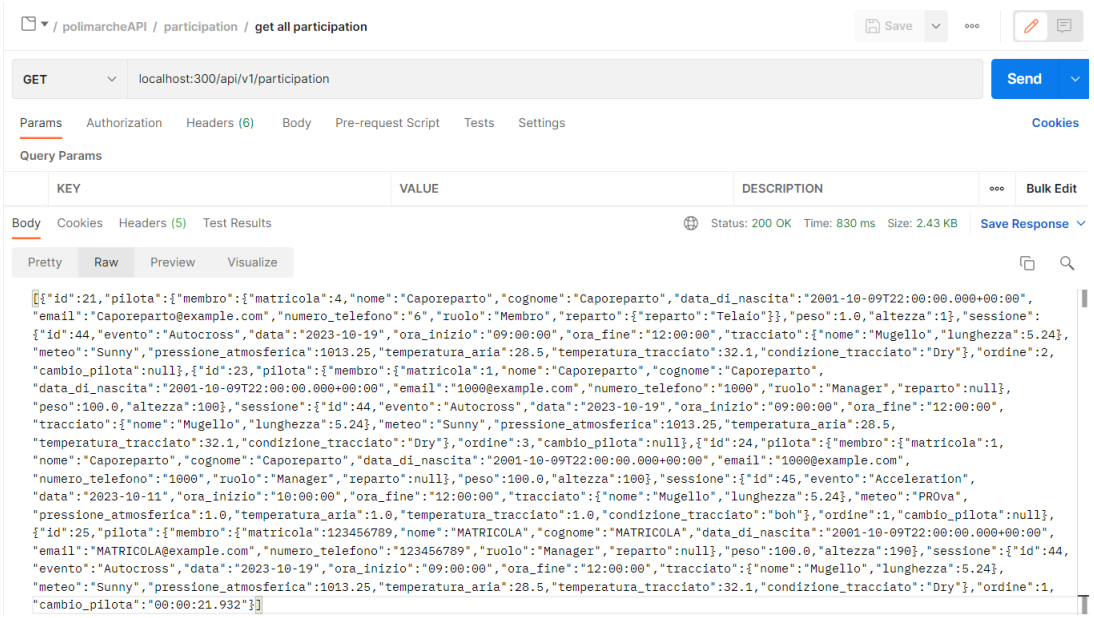


Figura 7.6: Test API - Request result

Capitolo 8

Conclusion

The application has been correctly developed satisfying all the requirements analyzed in Chapter 2.

As mentioned many times in this thesis, the software was thought to be further maintained and developed in the future. This approach allows for the addition of new functionalities and the simplification of existing ones to better fulfill the evolving needs of the application.

8.1 Future development

8.1.1 Utilization of a dedicated server

To enhance the efficiency of the mobile app in the future, the first step should involve setting up a dedicated server to host the already designed and implemented APIs. Additionally, this server can also serve as the foundation for a web application, meeting the team's needs in both mobile and web platforms.

8.1.2 Concurrency handling in the back-end system

The back-end software developed and analyzed in this thesis, just like the mobile app, should undergo further development. The initial implementation was not

designed to handle concurrent data access from multiple users, where requests for retrieving, creating, deleting, and modifying data from the server-hosted database could occur simultaneously.

8.1.3 Optimization of the database

The current database structure needs significant modifications to improve the efficiency of the API. This is especially crucial because the mobile application will enable users to store and view large volumes of telemetry data[1] collected from sensors during practice sessions.

Given the magnitude of data involved, it is essential to identify and address any potential bottlenecks in the database structure and optimize them for efficient data management.

[1]Know also as Big Data

Bibliography

[1] Rashedul Islam, Rofiqul Islam, and Tohidul Mazumder. Mobile application and its global impact. *International Journal of Engineering & Technology*, 10(6):72–78, 2010.

[2] Rachel Harrison, Derek Flood, and David Duce. Usability of mobile applications: literature review and rationale for a new usability model. *Journal of Interaction Science*, 1:1–16, 2013.

[3] Aakanksha Tashildar, Nisha Shah, Rushabh Gala, Trishul Giri, and Pranali Chavhan. Application development using flutter. *International Research Journal of Modernization in Engineering Technology and Science*, 2(8):1262–1266, 2020.

[4] Ghusoon Idan Arb and Kadhum Al-Majdi. A freights status management system based on dart and flutter programming language. In *Journal of Physics: Conference Series*, volume 1530, page 012020. IOP Publishing, 2020.

[5] Chunnu Khawas and Pritam Shah. Application of firebase in android app development-a study. *International Journal of Computer Applications*, 179(46):49–53, 2018.

[6] John D Blischak, Emily R Davenport, and Greg Wilson. A quick introduction to version control with git and github. *PLoS computational biology*, 12(1):e1004668, 2016.

[7] Philip A Bernstein and Umeshwar Dayal. An overview of repository technology. In *VLDB*, volume 94, pages 705–713, 1994.

[8] Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons, et al. Spring boot reference documentation. *Retrieved June*, 22:36, 2020.

[9] Siva Prasad Reddy Katamreddy and Sai Subramanyam Upadhyayula. Testing spring boot applications. In *Beginning Spring Boot 3: Build Dynamic Cloud-Native Java Applications and Microservices*, pages 295–323. Springer, 2022.

Ringraziamenti

In chiusura di questo lavoro, desidero esprimere la mia profonda gratitudine ai miei genitori, i quali, nonostante il mio carattere introverso, mi hanno sostenuto in ogni fase di questo percorso.

Ringrazio di cuore mia sorella Alessia, la cui vita e scelte sono diventate la mia fonte di ispirazione e il mio punto di riferimento. Grazie a lei ho appreso che con dedizione, impegno e notevoli sacrifici è possibile raggiungere ogni obiettivo che ci si prefissa.

Desidero inoltre esprimere il mio profondo ringraziamento a Sara, che durante l'ultimo anno è stata un grande sostegno per me, soprattutto a livello personale.

Infine, voglio estendere i miei ringraziamenti a tutti i compagni di corso con i quali ho condiviso le esperienze più significative durante tutti questi anni trascorsi insieme.