https://lib.uliege.be https://matheo.uliege.be

Development of a Flutter module for ATHLETin

Auteur : Alakhir, Ahmed
Promoteur(s) : Mathy, Laurent; Gain, Gaulthier
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2022-2023
URI/URL : http://hdl.handle.net/2268.2/17386

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

University of Liège - Faculty of Applied Sciences

Development of a Flutter module for *ATHLETin*

Student
Ahmed Alakhir

Promoter
Prof. Laurent Mathy

Thesis conducted for obtaining the Master's degree in Computer Science Engineering

Academic year 2022-2023

Abstract

Development of a Flutter module for *ATHLETin*

Ahmed ALAKHIR

Master in computer science engineering
University of Liège - Academic year 2022-2023

Promoter : **Prof. Laurent Mathy**

ATHLETin is a mobile and web app aiming to help the management of athletes in order to prevent injuries. The idea of such app emerged from Julien PAULUS, who believes that injuries could be avoided with better solutions to oversee athletes. This solution implies more communication between the athletes, the coaches and the medical specialists.

In practice, *ATHLETin* is divided into several modules that will handle specific parts of the solution such as a calendar module to ease the planning and a training module to manage training session properly. In this thesis, the administrator module will be implemented. Its role is to provide to authorized members an overview of the different data stores such as the list of athletes or their answers to specific questionnaires. To do so, a Flutter web-app will be implemented.

With the implementation of such module, challenges will appear, the main ones being the performance of the module, its flexibility and its integration to the existing architecture of the whole *ATHLETin* project. To address these, several techniques were used such as the following of architectural pattern, the pagination of the data or minimization the server requests. Moreover, the use of the Flutter technologies allow to have an optimized app by minimizing the screen refreshment using state management tools such as Provider. Eventually, the designed app will be deployed using a `Docker` container.

As this work only constitute a subpart of the global *ATHLETin* project developed by professor Laurent MATHY and his team, the module might be subject to further modification and improvements. This thesis will thus also play the role of documentation for the future developers working on *ATHLETin* modules.

Acknowledgements

Beforehand, I want to give special thanks to all the people who helped me throughout the completion of this thesis.

My gratitude goes toward Professor Laurent MATHY who allowed me to work on a thrilling subject which made me learn many concepts about fields I like in computer science. I am also grateful for his continuous supervision during the year.

I would also like to thank his team, especially Gaulthier Gain who was always keeping contact with me to ensure I had no problems. His availability and feedbacks really helped me during the completion of this work.

Obviously, the realization of this thesis would not have been possible without the different teachings of the professors of the faculty. Among these, I extend my appreciation to Prof. Guy LEDUC and Prof. Quentin LOUVEAUX for evaluating this work.

I want to conclude this acknowledgement session with a specific mention for my family who accompanied during these five years of study. I am truly grateful for their support which made me able to work under the best conditions.

Contents

I Introduction 1

1 Introduction 2

1.1 The creation of *ATHLETin* . . . 2
1.2 Use and functionalities . . . 3
1.2.1 Admin Module . . . 3
1.2.2 Training Module . . . 3
1.2.3 Agenda Module . . . 4
1.2.4 Medical Module . . . 4
1.2.5 Communication Module . . . 4
1.3 Challenges . . . 4

II Technical background 6

2 Backend tools 7

2.1 PostgreSQL . . . 7
2.2 Go . . . 7
2.3 GORM . . . 8
2.4 REST . . . 9

3 Frontend tools 11

3.1 Dart . . . 11
3.1.1 Type safety . . . 11
3.1.2 Garbage collection . . . 12
3.1.3 Null safety . . . 12
3.1.4 Asynchronous operations . . . 13
3.1.5 Libraries . . . 14
3.1.6 Platforms . . . 14
3.2 Flutter . . . 15

3.2.1 Architectural overview . . . 16
3.2.2 Widgets . . . 18
3.2.3 State management . . . 21
3.2.4 Hot reload . . . 24
3.2.5 Rendering process . . . 24
3.2.6 Navigation . . . 26

III Module implementation **29**

4 High level Architecture **30**

5 Components **32**
5.1 The database . . . 32
5.2 The REST server . . . 36
5.2.1 Description . . . 36
5.2.2 Architecture . . . 36
5.2.3 Authentication . . . 39
5.2.4 API . . . 39
5.3 The Web application . . . 41
5.4 The mobile application . . . 41

6 Administrator module **42**
6.1 Overview . . . 42
6.1.1 Questions management . . . 42
6.1.2 Results management . . . 43
6.2 Pages . . . 44
6.2.1 Login Page . . . 44
6.2.2 Front Page . . . 44
6.2.3 Try Later Page . . . 44
6.2.4 Studies Page . . . 44
6.2.5 Events Page . . . 46
6.2.6 Results Page . . . 49
6.2.7 Athletes Page . . . 51
6.2.8 Athletes/Study Page . . . 52
6.2.9 Members Page . . . 52
6.2.10 Profile Page . . . 53
6.2.11 Page routing . . . 53

7 Implementation structure **54**
7.1 Architectural Pattern . . . 54
7.1.1 Model View Controller pattern . . . 54
7.1.2 Model View View-Model pattern . . . 56
7.2 Services . . . 58
7.2.1 Navigation service . . . 58
7.2.2 Identity service . . . 58
7.2.3 Network service . . . 58
7.2.4 Provider Service . . . 59
7.3 Files organization . . . 61
7.4 Performance . . . 61
7.4.1 Caching . . . 61
7.4.2 Pagination . . . 62
7.4.3 Searching . . . 63
7.4.4 Measuring the performance . . . 64

8 Testing **67**
8.1 Unit Tests . . . 67
8.2 Widget Tests . . . 67
8.3 Integration Tests . . . 68
8.4 Manual Tests . . . 68

9 Deployment **69**
9.1 Modules encapsulating . . . 69
9.2 Reverse Proxy role . . . 70

IV Conclusion **71**

10 Conclusion **72**

11 Future work **74**
11.1 *ATHLETin* . . . 74
11.1.1 Session persistence . . . 74
11.1.2 More languages . . . 74
11.1.3 Role Management . . . 74
11.2 Admin Module . . . 75
11.2.1 Question creation . . . 75
11.2.2 Event Visualization . . . 75

11.2.3 Improve Performances 75
11.2.4 Adding features 75

A Module screenshots **76**

Bibliography **90**

Part I

Introduction

Chapter 1

Introduction

Before going through the technical parts, a first description of the project's context is necessary. This introduction will go through the story of *ATHLETin* from its creation to the current use. This will be followed by the current challenges that lead to this project.

1.1 The creation of *ATHLETin*

Officially created in 2020, *ATHLETin*'s origin are actually 10 years old. Its concept emerged from Julien Paulus, physical trainer of the Sart-Tilman's rugby training center. The problem faced was simple : as athletes were training both in the training center and with their respective team, communication between these entities was necessary for a better management of their training session.

The first solution, developed by a co-worker of Julien Paulus, was *MyLBFR*, a web application handling this communication. It managed to solve the issue and started to add more features to become more complete. Eventually, the app was cancelled due to a lack of investment.

However, this was not felt as a back to square one for Mr Paulus. Indeed, as the app made its proof by solving the communication problem and improved the management of training, he knew that this type of platform was the right solution. This will to design a better platform will lead him to Xavier Picard, specialist in management and marketing of sports organizations. With him, they will start a collaboration with Laurent Mathy, professor at Liège University, for the development of what will become the current version of *ATHLETin*.

This leads to the current state of the project, the app is still being developed and maintained by Laurent Mathy and his team, which includes students.

1.2 Use and functionalities

ATHLETin benefits from being the second version of an athlete management app, this allow the app to features numerous functionalities that are known as useful for this management task. These include more targeted fields of the athlete's management such as the follow-up of injuries or the agenda. All these tasks are separated in different modules [1].

1.2.1 Admin Module

As its name suggests, this module will only be used by administrators. It must be able to communicate with the different part of the web app. The admins can :

- interact with the calendar
- generate convocations for athletes
- manage training logs and attendance records
- communicates with the athletes
- export data as pdf, Excel or csv

It is also possible to personalize the different authorization. **This module is the one implemented for this thesis.**

1.2.2 Training Module

As said in 1.1, the initial goal of the app was to manage the training. It is thus a very important part of the app. All this management is based on processing data collected directly by asking athletes to answer questionnaires at every training session. The processed uses semi-automatic algorithms to analyze the results. Athlete's performances are regrouped there and can be updated and consulted for more detailed analysis.

1.2.3 Agenda Module

The different events are listed on a calendar with all their details. It allows a global view on competitions, internships, training or treatment sessions and medical consultations. It also checks if the player attends sessions or if he's absent [2].

1.2.4 Medical Module

All the health tracking of athletes will be handled by this module. Injuries will be precisely described to get the best possible diagnostics through time. The integration of this module in the app can allow quick access to injuries specificity such as the context of the injury when the date is known. It also allows to know what events a sportsman will miss when he's injured. Medical professionals will play a major role as they will not only handle the diagnostics but also communicate with the training coaches. The platform will also directly allow them to make appointments with the players [2].

1.2.5 Communication Module

This simply provides a messaging service between all the members.

1.3 Challenges

The numerous functionalities will require a good design to keep the application simple and ergonomic. As the app can be used in different context, it needs to support multiples platforms. This is why Laurent Mathy and his team chose to use Dart and Flutter for the development of the application. The other technical choices were motivated by the ease of use for the development team to allow a quality maintenance. The team decided to use a `REST` server implemented in Go and a PostgreSQL database. More details about the architecture will be provided in the following chapters.

Each module is developed separately into web platforms. On top of that, a mobile app was created for the athletes to manage their agenda and questionnaires. During development, some choices made at the beginning need to be revisited for better compatibility and scale handling. Notably, the structure of the database has been revisited to allow more flexibility. One of the challenges covered by this thesis is the reimplementation of the administrator module in order to take in account the

different changes as well as keeping this implementation flexible enough for further changes.

> This introduction's goal was to summarize the general context of this project to enlight its objectives. Before going through the different tasks applied during this thesis, a chapter will be dedicated to the technical details of the app in order to get familiar with the different tools used.

Part II

Technical background

Chapter 2

Backend tools

2.1 PostgreSQL

App managing data about users will obviously require a database to store its data. The relational database model presents itself as the most suitable for this management. There exist many database management system dedicated to relational databases, the main requirements to handle data from Athletin are :

- Free and open-source : To avoid unnecessary costs for support and maintenance.
- Advanced security features : Medical data can be sensitive so it is important to guarantee their confidentiality.
- SQL compliant : SQL is one of the most used language and shows many advantages to handle structured data.
- Scaling: The system must be able to handle complex queries and large amounts of data.

The features proposed by PostgreSQL [3] correspond more to the needs of Athletin, which is why this system was chosen.

2.2 Go

Go [4], also known as Golang, is a statically typed, compiled, concurrent, garbage-collected programming language. Designed in 2007 by Google, it was created to

handle large scale networked systems and support concurrency and parallelism. Indeed, Robert Griesemer, Rob Pike, and Ken Thompson, the 3 designers of Go wanted to benefits from the safety and performance of statically typed compiled languages (Java, C++) with the expressiveness and convenience of a dynamically typed interpreted language (Ruby, Python, JavaScript) [5].

Some of the main benefits of Go include:

- Simple and Easy to Learn: Clean and simple syntax, the language is thus easy to learn.
- Concurrency Support: Concurrency is supported through the use of Goroutines and channels, which make it easier to write and maintain concurrent systems.
- High Performance: Go is designed to be fast and efficient, and can handle large amounts of data and traffic.
- Memory safety : In Go, variables are zero-initialized, indexing is bounds-checked, it also uses pointer in a safer way than other languages.
- Garbage Collection: Go uses a garbage collector to manage memory automatically, reducing the risk of memory leaks and other related problems.
- Statically Typed: The language is statically typed, which can help prevent many types of runtime errors.
- Cross-platform: Its code can run on any platform that supports the language, making it an ideal choice for building cross-platform applications.

Overall, Go offers a balanced mix of simplicity, efficiency, and power, making it an attractive choice for many types of projects and applications. Another reason motivating the choice of Go is the GORM framework aiming to simplify the database interactions.

2.3 GORM

GORM (Golang ORM library) [6] is an Object Relational Mapping (ORM) library for the Go programming language. It provides a simple way to interact with databases, allowing to work with databases using Go code instead of writing raw SQL queries.

The main advantages of using GORM are:

- Abstraction: GORM provides a higher level of abstraction over databases.
- Productivity: The simplicity of the interactions allow a faster and more efficient development.
- Ease of use: The API is simple and intuitive.
- Active development: GORM is open-source and the community is active, libraries are constantly being improved and bugs are fixed quickly.

Moreover, GORM benefits from using models. These are data structures that follow some conventions allowing an easy storage in the database.

2.4 REST

A RESTful API [7] is a type of web service that follows the principles of the REST (Representational State Transfer) architectural style. This means that it is designed to provide a standardized way for different computer systems to communicate with each other over the internet.

The REST architecture is based on a set of constraints that define how resources are identified and addressed, and how interactions between clients and servers are handled. These constraints include:

- Client-server architecture: The client and server are separated from each other, with each responsible for a specific set of tasks.
- Stateless communication: Each request from the client to the server must contain all the information needed to complete the request. The server does not keep any information about the client or the previous requests.
- Uniform interface: The API must provide a consistent way of accessing and manipulating resources, with standard methods and representations for data.
- Cacheable: Responses from the server must be marked as cacheable or non-cacheable to help reduce the number of requests sent to the server.
- Layered system: A client should not be able to tell whether it is communicating directly with the server or with an intermediary, such as a load balancer.

To build a RESTful API, developers must define a set of resources that can be accessed by clients. Each resource should have a unique identifier and should be represented in a standard format, such as JSON or XML. The API must also provide a set of methods for accessing and manipulating these resources, such as GET, POST, PUT, and DELETE.

Clients can then use these methods to interact with the resources provided by the API, sending requests to the server and receiving responses in the specified format. Responses from the server must include a status code that indicates whether the request was successful or not, along with any relevant data or metadata.

Overall, a RESTful API provides a flexible and standardized way for different computer systems to communicate over the internet, enabling developers to create scalable, reliable, and easily maintainable web services.

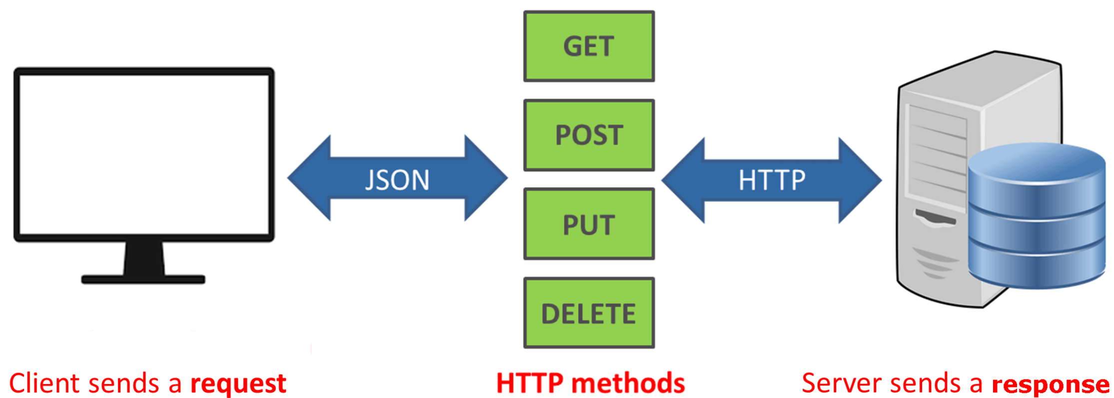


Figure 2.1: REST API architecture

Chapter 3

Frontend tools

3.1 Dart

Created by Google in 2011, Dart [8] is presented as a *"client-optimized language for developing fast apps on any platform"* [9]. It's an open-source, structured programming language designed for multi-platform development.

3.1.1 Type safety

Type safety in Dart [10] refers to the language's ability to catch type errors at compile-time rather than runtime. This is achieved by using a type system that ensures variables, parameters, and return values have consistent types throughout the code.

Dart's type system is similar to Java's in that it supports both static typing and dynamic typing. However, Dart allows developers to omit types and rely on type inference, which makes code more concise.

In comparison to JavaScript, which is dynamically typed, Dart's type safety provides better code reliability and can catch many errors before the code is executed. With dynamic typing, type errors can go unnoticed until runtime, which can cause unexpected behavior and crashes.

3.1.2 Garbage collection

Garbage collection [11] is automatically handled by the Dart runtime. Dart uses a generational garbage collector that divides objects into two generations: young and old. Most objects are created in the young generation, and the garbage collector periodically collects and frees the memory used by objects that are no longer used. The old generation contains long-lived objects so the process occurs less frequently.

In contrast to Java, which uses both generational and concurrent garbage collection, Dart's garbage collector [12] is not concurrent and may cause short pauses in the application during garbage collection. However, Dart's garbage collector is optimized for high performance and low latency, and its simplicity makes it suitable for many types of applications. Indeed, the garbage collector is designed to receive an alert when no user interaction happens and the app is idle in order to manage the memory without impacting the performance.

3.1.3 Null safety

Dart is a programming language that supports null safety, which means it helps developers avoid null reference errors in their code. Null safety in Dart [13] is achieved through the use of non-nullable types and optional types. By using these types, Dart's null safety feature ensures that variables are always initialized with a non-null value, and that null values cannot be assigned to variables that are not intended to hold them.

The benefits of Dart's null safety feature [14] include:

- Improved code reliability: With null safety, variables can be ensured to be initialized with non-null values, which reduces the risk of null reference errors that can cause application crashes or data corruption.
- Enhanced code readability: Differentiating non-nullable types and optional types already gives a piece of information about the code without having to add more documentation.
- Better code maintainability: With null safety, developers can make changes to their code with greater confidence, because they know that null reference errors are less likely to occur. This reduces the time and effort required to maintain and update code.

- Code optimization : The Dart compiler [15] will be able to optimise the code.

In summary, Dart's null safety concept helps developers write more reliable, readable, maintainable, and productive code by minimizing the risk of null reference errors.

3.1.4 Asynchronous operations

In Dart, asynchronous operations are handled using `futures` and `async/await` syntax [16]. `Futures` represent a value that may not be available yet. They allow developers to perform asynchronous operations, such as network requests or file I/O, without blocking the main thread of the application. Instead, the main thread can continue executing while the asynchronous operation completes in the background. When the operation completes, the future resolves with a value.

`Async/await` syntax provides a way to work with futures in a more readable and intuitive way. With `async/await`, developers can write asynchronous code that looks similar to synchronous code. The '`async`' keyword is used to declare a function that returns a `Future`, and the '`await`' keyword is used to wait for the result of a `Future`.

Here is an example of how asynchronous operations work in Dart:

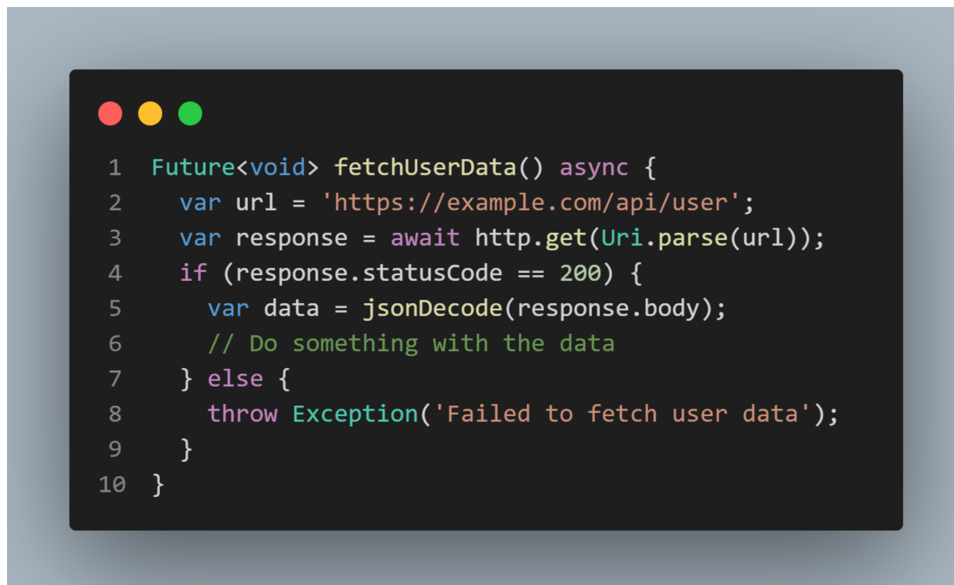


Figure 3.1: Example of Dart code using asynchronous operation

In this example, the `fetchUserData()` function returns a Future that completes with a void value. The `async` keyword is used to indicate that the function contains asynchronous code.

The function first defines a URL for a network request. Then it uses the http library to make a GET request to the URL, and waits for the response to complete using the `await` keyword. Once the response is available, it checks the response status code and either processes the data or throws an exception if the request failed.

By using Futures and `async/await` syntax, Dart provides a powerful and easy-to-use mechanism for working with asynchronous operations.

3.1.5 Libraries

Dart includes a large set of core libraries [17] that provide fundamental tools for web and mobile app development, including HTTP networking, asynchronous programming, and collections. It also has a modular architecture that enables developers to use only the libraries they need, reducing the size of the application.

3.1.6 Platforms

Dart's compiler technology allows code to be run in different ways. For mobile and desktop apps, Dart includes both a Dart VM with just-in-time (JIT) compilation and an ahead-of-time (AOT) compiler that produces machine code [9]. During development, the JIT compilation, together with incremental compilation, enables hot-reloading, allowing developers to see the changes they make in real-time. Once the app is released, the Dart AOT compiler produces native ARM or x64 machine code, providing better performance and consistent, shorter startup times.

For web apps, Dart includes both a development-time compiler (dartdevc) and a production-time compiler (dart2js). Both compilers translate Dart into JavaScript, which is then run in the browser. The dartdevc compiler supports incremental compilation and emits modular JavaScript, enabling a fast developer cycle. On the other hand, the dart2js compiler aims to compile Dart code into fast, compact, and deployable JavaScript, using techniques such as dead-code elimination.

Regardless of the platform or compilation method used, executing Dart code requires a Dart VM that provides the Dart runtime, which manages memory through a garbage collector policy. The Dart runtime is automatically included in self-contained executables on native platforms. Although the name "Dart VM" suggests that Dart is always interpreted or JIT-compiled, it can also be compiled ahead-of-time to native machine code, providing excellent performance.

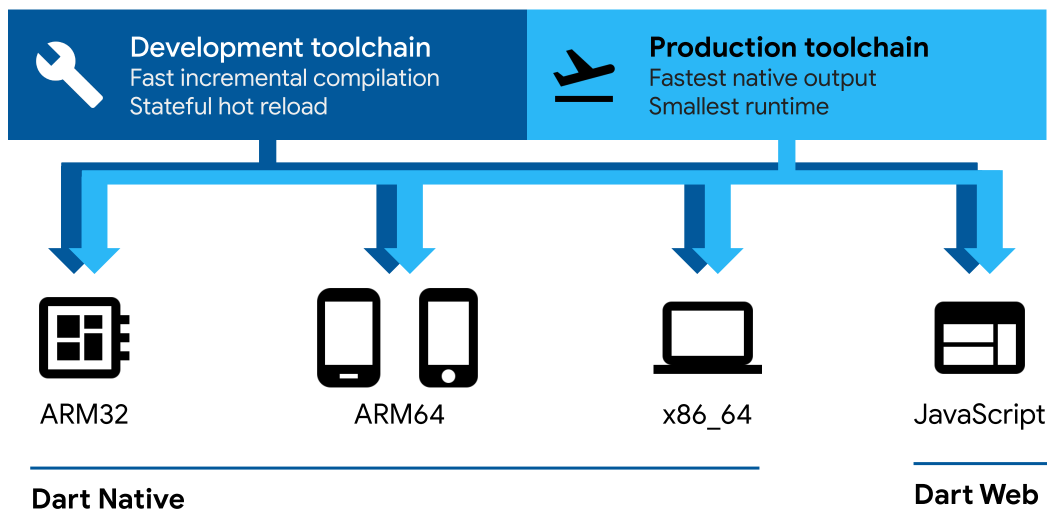


Figure 3.2: Dart platforms [9]

3.2 Flutter

The development of technology has led to the rise of different platforms for computing, including personal computers, the web, and mobile devices. With this increase in platform diversity, the need for efficient and effective ways to create cross-platform applications has become increasingly important. This is why in 2017, Google created Flutter [18], a modern, open-source, and free UI software development kit (SDK).

Flutter allows to build beautiful, fast, and high-performance applications for all platforms, including iOS and Android, as well as desktop, web, and embedded devices. With its fast development cycle and attractive, customizable widgets, Flutter has quickly become one of the most popular frameworks for building cross-platform applications.

3.2.1 Architectural overview

Flutter uses a layered architecture [19] that separates the UI from the business logic, allowing to focus on building the UI during the development while the framework handles the underlying platform differences. This architecture also enables Flutter to provide a consistent set of widgets across all platforms, which helps developers create applications that look and feel native on each platform.

The framework

The interaction with Flutter is usually performed through the Flutter framework, which is a modern, reactive framework written in Dart. The Flutter framework is relatively small and comprises various foundational libraries that can be extended using packages. These packages can add extra features and functionality such as platform plugins, web support, and animations. The framework comprises several components, including a platform-specific embedder, which provides an entry point and coordinates with the operating system to access necessary services such as rendering surfaces, accessibility, and input.

The engine

At the core of Flutter is the Flutter engine, which is mostly written in C++ and provides the essential features and functionality required for all Flutter applications. This engine is exposed to the Flutter framework through the dart:ui library, which wraps the underlying C++ code in Dart classes. The engine provides a portable runtime for executing Dart code, a set of low-level APIs for rendering graphics and text, and a set of platform plugins that provide access to platform-specific services such as location, camera, and sensors.

The embedder

The embedder is a platform-specific library that provides an entry point for the Flutter application and coordinates with the operating system to access necessary services such as rendering surfaces, accessibility, and input. The embedder also provides a platform-specific implementation of the Flutter engine, which is used to execute Dart code and render graphics and text. The implementation is written in Java and C++ for Android, Objective-C/Objective-C++ for iOS and macOS, and in C++ for Windows and Linux.

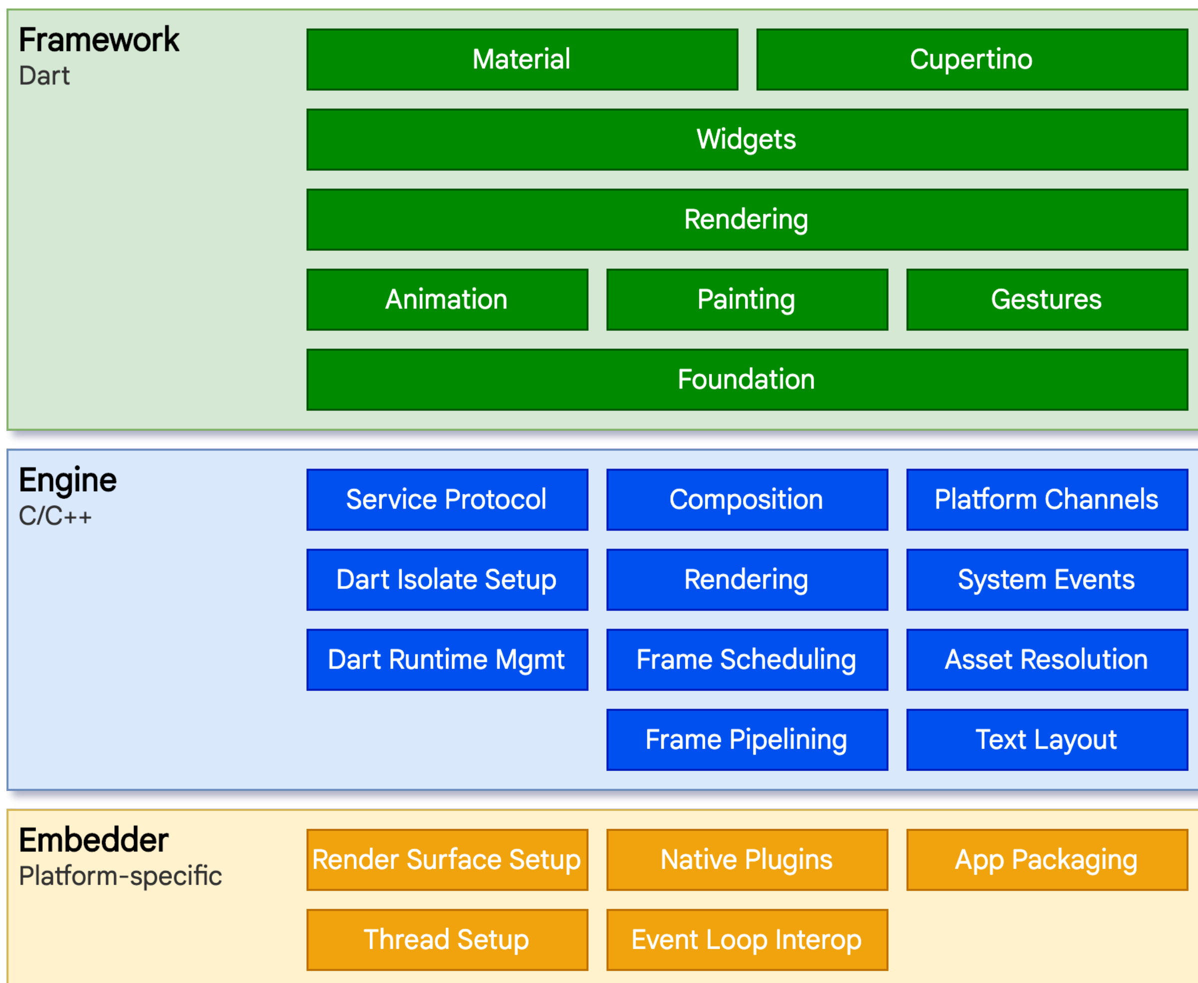


Figure 3.3: Flutter architectural layers [19]

Web support

While the general architectural concepts of Flutter apply to all platforms, some characteristics are specific to Flutter's web support. Dart has been compiling to JavaScript for a long time and Flutter is written in Dart, so it is relatively easy to compile it to JavaScript.

However, the Flutter engine is designed to interface with the underlying operating system and not the web browser. To solve this problem, Flutter provides a reimplementation of the engine on top of standard browser APIs to render Flutter content on the web. Currently, there are two options for rendering Flutter content on the web: HTML and WebGL.

In HTML mode, Flutter uses HTML, CSS, Canvas, and SVG to render content on the web, this mode offers the best code size characteristics. With WebGL, Flutter uses a version of Skia compiled to WebAssembly called CanvasKit. This version provides the fastest path to the browser's graphics stack and leads to higher graphical fidelity with the native mobile targets.

Unlike other platforms on which Flutter runs, there is no need for Flutter to provide a Dart runtime on the web. Instead, the Flutter framework, along with any code written, is compiled to JavaScript.

During development, Flutter web uses `dartdevc`, a compiler that supports incremental compilation and allows hot restart for apps. Conversely, when developers are ready to create a production app for the web, they use `dart2js`, a highly-optimized production JavaScript compiler that packages the Flutter core and framework along with the application into a minified source file that can be deployed to any web server.

3.2.2 Widgets

Flutter's widgets are the building blocks of the UI in a Flutter app [20]. Widgets represent the various components of the UI, such as buttons, text inputs, and containers, and can be organized into trees to create complex and dynamic layouts. Widgets are defined as classes in Dart, and each widget has a build() method that returns a description of the UI elements that should be displayed on the screen.

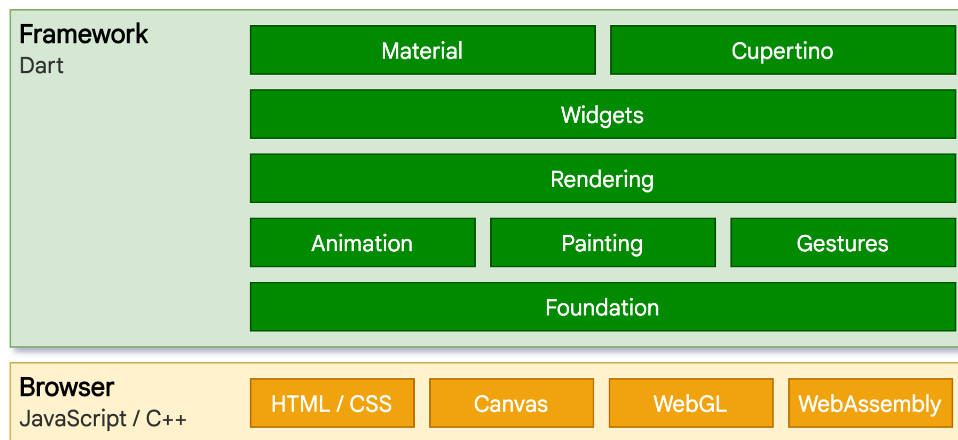


Figure 3.4: Flutter architecture for web [19]

All widgets in Flutter are organized in a tree structure. At the root of the tree is typically a `MaterialApp` or `CupertinoApp`, which represent the starting point of the application. Underneath this root widget, widgets are added to create the UI. These nested widgets create a hierarchical structure known as the widget tree.

Stateless and stateful widgets

A `StatelessWidget` [21] is a widget that does not have any mutable state. It takes in some data as input, performs some calculations, and renders the result. This means that a `StatelessWidget` is immutable, meaning it cannot change after being created. Because `StatelessWidgets` are immutable, they can be easily reused, which is an important characteristic of well-structured Flutter code.
On the other hand, a `StatefulWidget` [22] is a widget that has mutable state. This means that it can change its appearance based on data that changes over time, such as user input or network data. A `StatefulWidget` consists of two classes: the widget class and the state class. The widget class is immutable, while the state class is mutable.
When a `StatefulWidget` is created, it creates an instance of the associated state class. The state is then used to manage mutable state and provide a build method, which is responsible for describing how the widget should look based on the current state. When the state changes, the `StatefulWidget` calls the `setState()` method of its associated state object. This will rebuild the widget tree with the new state values, effectively updating the UI to reflect the new data.
In addition to the **build()** method, `StatefulWidget` also has a `dispose()` method

that is called when the widget is removed from the tree. This will clean up any resources that the widget has allocated, such as stopping animations or closing network connections.
The lifecycle of a *StatefulWidget* [23] goes through several stages:

1. `createState()` [24] is called when the widget is first created, and it returns an instance of the associated state class.

2. `initState()` [25] is called once when the state is initialized, and is often used to perform initialization that is required for the widget to function properly.

3. `build()` [26] is called whenever the widget needs to be rebuilt, which can occur multiple times during the lifetime of the widget.

4. `setState()` [27] is called when the widget's state has changed and it needs to be rebuilt.

5. `didUpdateWidget()` [28] is called after a widget has been rebuilt with new data. It is used to handle any actions or updates that are necessary after the rebuild.

6. `dispose()` [29] is called when the widget is removed from the tree, and is used to clean up any resources that the widget has allocated.

Overall, widgets are a fundamental part of building UIs in Flutter, and understanding the differences between stateless and stateful widgets is essential for creating dynamic and responsive user interfaces. By using the appropriate widget for each part of an app's UI, , developers can create complex UIs that are both performant and easy to maintain.

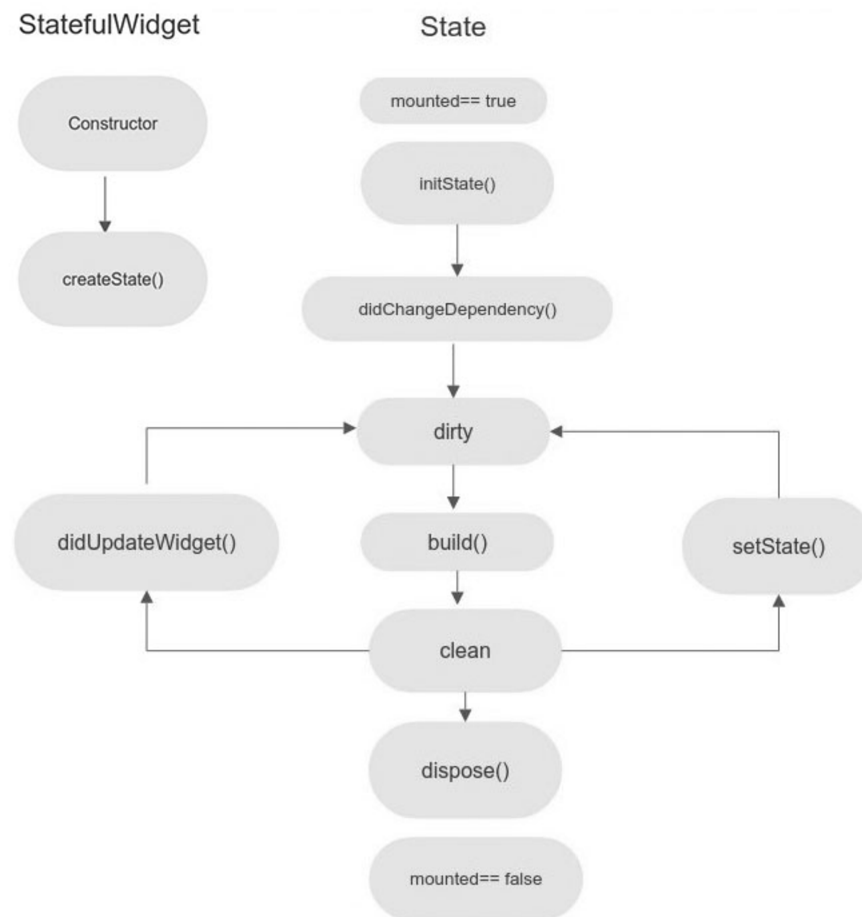


Figure 3.5: State lifecycle of a stateful widget [30]

3.2.3 State management

After covering the basics of stateful and stateless widgets, the concept of state management in Flutter needs to be explored in detail. As seen earlier, stateful widgets can maintain their own state, while stateless widgets cannot. However, managing the state of an application can become complex and lead to code that is difficult to maintain and scale, which is where state management comes in. It provides a way to manage and update state across the app.

InheritedWidget

`InheritedWidget` [31] is a popular approach to state management in Flutter that allows data to be passed down the widget tree, from parent widgets to their children. `InheritedWidget` defines a piece of data as an object that can be accessed by any child widget, and the widget tree is then rebuilt whenever this data changes, ensuring that all widgets using that data are updated accordingly. This approach can simplify state management by allowing multiple widgets to access the same data without needing to pass it explicitly through their constructors, and it can improve performance by minimizing the number of widget rebuilds necessary when data changes.

To understand `InheritedWidget` better, it is important to take a look at the widget tree. In Flutter, every widget has a parent and zero or more children, forming a tree-like structure where the root is the top-most widget and the leaves are the lowest-level widgets. When an `InheritedWidget` is added to a parent widget, its data becomes available to all of its descendants in the widget tree. Any descendant that depends on this data will automatically rebuild when the data changes, thereby updating multiple widgets across the app that depend on it while minimizing unnecessary widget rebuilds.

Provider

Provider[32] is a state management package for Flutter that wraps around the `InheritedWidget` class. It provides a simple and scalable way to manage app state by using a combination of dependency injection and a data stream that notifies the app when the state has changed.

The core concept of Provider is the concept of a provider, which is simply a widget that exposes a piece of data to its descendants. This data can be any object or value that the app needs to manage state, such as a user's profile information, a list of items to display, or a theme object. By using providers, widgets can easily access and update this data without needing to pass it down manually through constructors.

Providers are defined by creating a new class that extends the `ChangeNotifier` class [33]. This class represents the data that the provider will expose, and it contains any necessary business logic for updating that data. When the data changes, the provider calls its `notifyListeners()` [34] method to tell all of its descendants that the data has been updated.

To access the data from a provider, widgets can use the `Provider.of()` method to obtain an instance of the provider. This method looks up the widget tree for the nearest provider of the specified type, and returns the data exposed by that provider. When the data changes, any widget that depends on it will be automatically rebuilt,

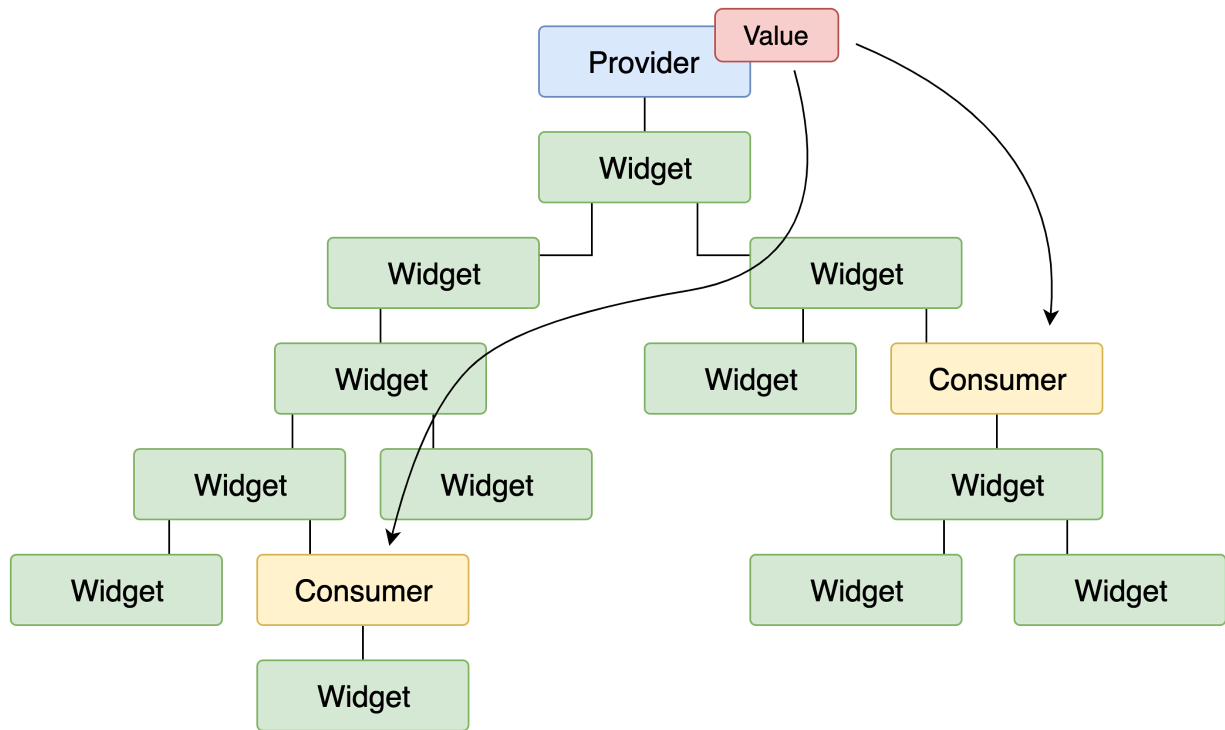


Figure 3.6: Widget tree using Provider [35]

thanks to the `notifyListeners()` method.

One of the strengths of Provider is its flexibility. Providers can be defined at any point in the widget tree, and any widget that is a descendant of that provider can access its data. This makes it easy to define providers that encapsulate different parts of the app's state and business logic [36]. For example, one provider might manage the user's authentication status, while another might manage a list of items that the user has requested.

Provider also supports a variety of advanced features, such as lazy initialization of data, providing different instances of a provider based on context, and scoping providers to specific parts of the widget tree.

3.2.4 Hot reload

Hot Reload [37] is a powerful feature in Flutter allowing to quickly see the effects of code changes without having to stop the app and restart it from scratch. This feature speeds up the development process as it allows to experiment with different code changes, quickly iterate on designs, and fix bugs on the fly.

The Hot Reload system in Flutter works by compiling only the code that has changed, and then injecting it directly into the running Dart VM [38]. This means that the effects of the code changes can be seen almost instantly, without having to wait for the app to fully rebuild and restart. This can be especially useful for debugging and testing, as it allows developers to quickly identify and fix issues in their code.

One of the key benefits of Hot Reload is that it allows to work in a more iterative and experimental way. Indeed, seeing the results of code changes immediately, without having to worry about breaking the app or causing other issues, will help to streamline the development process and make it more efficient.

However, there are some limitations to the Hot Reload system in Flutter. For example, it may not work well for large codebases or complex applications, where rebuilding and reloading can take longer. In addition, Hot Reload may not be suitable for certain types of changes, such as changes to the app's architecture or state management system [39].

Despite these limitations, Hot Reload remains a powerful feature in Flutter, helping to create apps in less time and with fewer errors.

3.2.5 Rendering process

When a Flutter application is launched, it creates a widget tree, where each widget is an immutable declaration of a part of the UI. When the UI needs to be rendered, Flutter calls the `build()` method of each widget, which returns a subtree of widgets that render the UI based on the current app state. During this process, the `build()` method can introduce new widgets, as necessary, based on its state.

Flutter translates the widgets expressed in code into a corresponding element tree during the build phase, with one element for every widget. Each element represents a specific instance of a widget in a given location of the tree hierarchy. There are two basic types of elements:

1. `ComponentElement` : host for other elements
2. `RenderObjectElement` : element that participates in the layout or paint phases

`RenderObjectElements` are an intermediary between their widget analog and the underlying `RenderObject` .

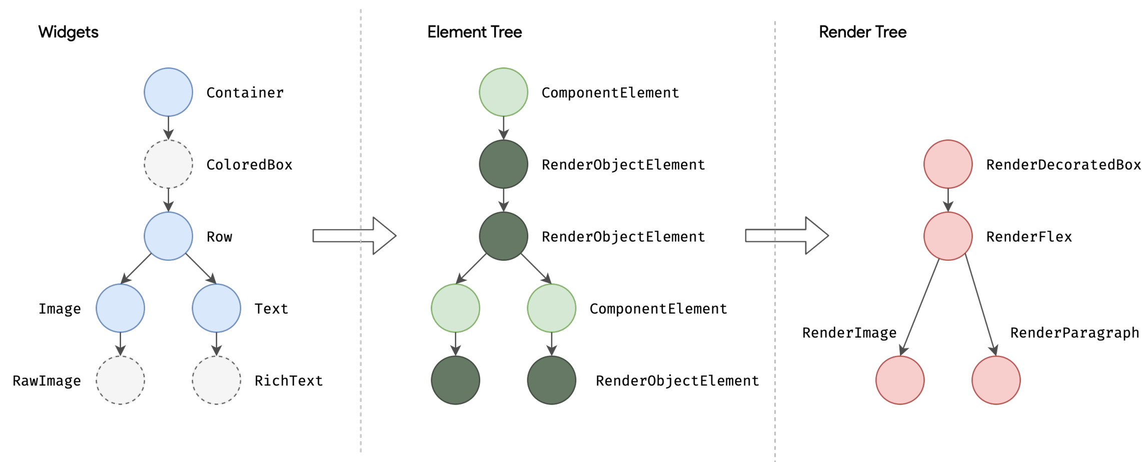


Figure 3.7: Widget tree rendering [19]

The element for any widget can be referenced through its `BuildContext` , which is a handle to the location of a widget in the tree. This is the `context` in a function call such as `Theme.of(context)` , and is supplied to the `build()` method as a parameter.

Flutter's widget tree is persistent from frame to frame, and therefore plays a critical performance role, allowing Flutter to act as if the widget hierarchy is fully disposable while caching its underlying representation. By only walking through the widgets that changed, Flutter can rebuild just the parts of the element tree that require reconfiguration.

Flutter uses a box constraint model to efficiently lay out a hierarchy of widgets. The base class for every node in the render tree is `RenderObject` , which defines an abstract model for layout and painting. During the build phase, Flutter creates or updates an object that inherits from `RenderObject` for each `RenderObjectElement` in the element tree. Most Flutter widgets are rendered by an object that inherits from the `RenderBox` subclass, which represents a `RenderObject` of fixed size in a 2D Cartesian space. To perform layout, Flutter walks the render tree in a depth-first traversal and passes down size constraints from parent to child. In determining its size, the child must respect the constraints given to it by its parent. Children respond by passing up a size to their parent object within the constraints the parent established.

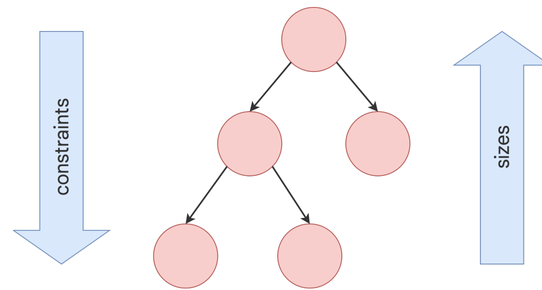


Figure 3.8: Constraints and sizes propagation in the widget tree

The root of all `RenderObjects` is the `RenderView` , which represents the total output of the render tree. When the platform demands a new frame to be rendered, a call is made to the `compositeFrame()` method, which is part of the `RenderView` .

The `compositeFrame()` method is responsible for generating a bitmap of the current frame by walking the render tree and issuing a call to each `RenderObject`'s `paint()` method. `paint()` is responsible for issuing the appropriate draw commands for rendering to the underlying graphics API, such as OpenGL or Metal.

After all the `RenderObjects` have been painted, the resulting bitmap is passed to the compositor, which composites the bitmap with any other layers that are part of the view hierarchy (such as a navigation bar or a status bar), applies any necessary effects (such as shadows or blurs), and finally displays the result on the screen.

In summary, Flutter's rendering engine provides a performant and flexible way to render complex UIs by using a widget tree and a render tree. The widget tree is immutable and defines the structure of the UI, while the render tree is mutable and handles layout and rendering. By passing constraints down from parent to child, Flutter can efficiently layout objects in $O(n)$ time, and by caching the element tree from frame to frame, it can rebuild just the parts that require reconfiguration. Finally, by using the `RenderObject` class and its subclasses, Flutter can handle a wide range of use cases and easily interface with underlying graphics APIs to provide a smooth and performant user experience.

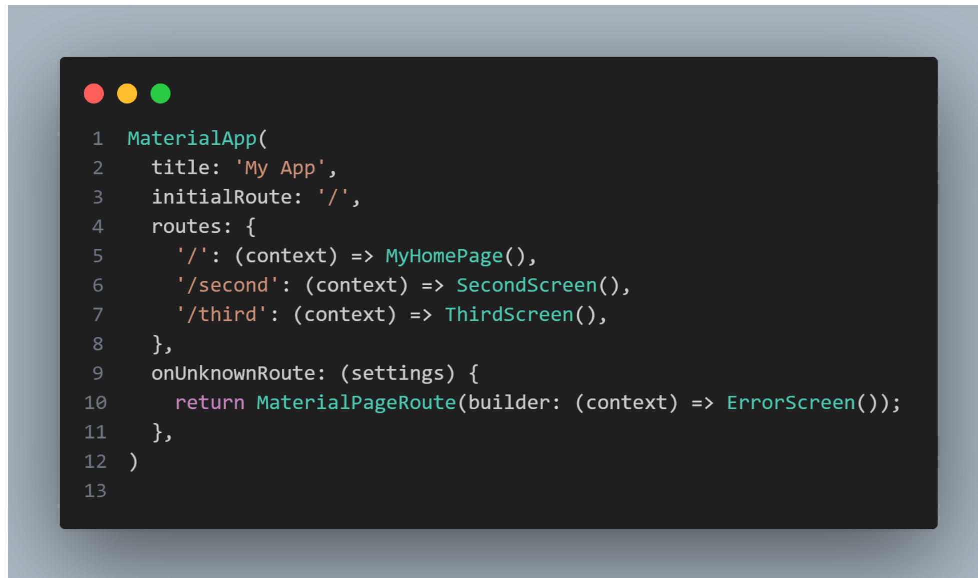
3.2.6 Navigation

Flutter provides a powerful and flexible navigation system for moving between different screens and views in an application [40].

One key feature of Flutter's navigation system is the concept of named routes. Named routes allow to navigate to a specific screen or view in the application using

a predefined name. This makes it easier to manage and organize the application's navigation structure and helps to keep the code clean and maintainable.

To use named routes in a Flutter application, they first need to be defined them in the application's `main.dart` file. The MaterialApp widget provides a namedRoutes parameter to define the routes.



```
MaterialApp(
  title: 'My App',
  initialRoute: '/',
  routes: {
    '/': (context) => MyHomePage(),
    '/second': (context) => SecondScreen(),
    '/third': (context) => ThirdScreen(),
  },
  onUnknownRoute: (settings) {
    return MaterialPageRoute(builder: (context) => ErrorScreen());
  },
)
```

Figure 3.9: Example of routes definition

The Navigator widget's pushNamed method allows to navigate to a named route. This method takes two arguments: the BuildContext and the name of the route to navigate to. Other arguments can be added in order to pass data from a page to another. This method will also allow to easily handle unknown routes by redirecting to a specific error page instead of a blank screen as set by default by Flutter.

In this chapter, the selected tools for implementing the REST server were presented, with a focus on maintainability and efficiency. The Go language was chosen for its simplicity and performance, along with the integration of PostgreSQL as the database technology. Furthermore, a comprehensive exploration of the Dart language and the Flutter framework was provided, offering insights into these frontend tools.

It is important to note that while the explanations provided in this chapter offer a solid foundation, the presented overview of these tools is not exhaustive. Both Go and Flutter are vast and complex ecosystems with numerous features and capabilities. However, the information covered so far should be sufficient to grasp the upcoming section that will review the implementation details of the project.

The next part will cover the implemented work for this project, starting with an overview of the architecture to better understand its goals. It will then delve into the main components, providing important details about the parts of the architecture used. Finally, a detailed description of the implemented module will be presented.

Part III

Module implementation

Chapter 4

High level Architecture

Before diving into the implementation of the module, a description of the global architecture is necessary. Indeed, the module built for this thesis is itself a part of the complete *ATHLETin* project. By examining the architecture, valuable insights can be gained into how the various elements work together to achieve the desired functionality and meet the project's objectives.

Currently, the architecture allows to gather information relative to athletes under the care of trainers/doctors. The structure is composed by :

1. a REST server
2. a secure database
3. a web-based administration application
4. a mobile application

Figure 4.1 illustrates the architecture of the system and depicts the various components involved, along with their interactions. The REST server (1) serves as the intermediary between the other components and the database (2). It acts as the sole entity responsible for direct communication with the database, meaning that any component wishing to access stored information must go through the REST server. This design choice enhances security by centralizing and controlling the communication with the database.

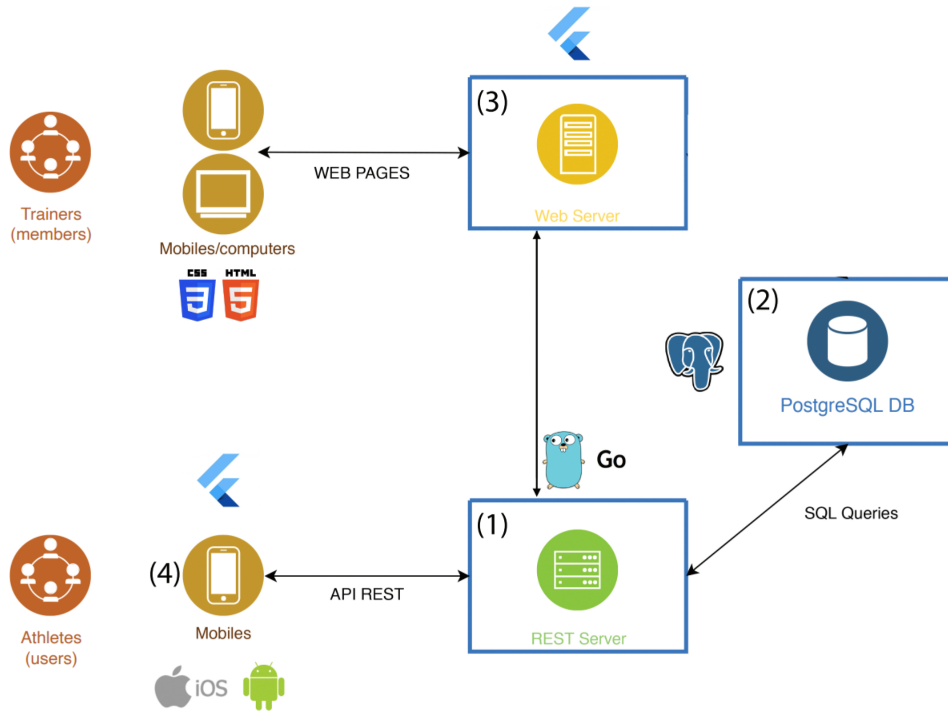


Figure 4.1: Interactions between the different components of the IT architecture [41]

Two components establish network communication with the REST server: the administration server (3) and the mobile application (4). Both components utilize JSON format for performing HTTP requests to retrieve and transmit data to the REST server. Once the information is stored in the system, trainers and doctors can utilize the administration interface to export data pertaining to individual athletes or a group of athletes. This functionality enables convenient data exportation for analysis and reporting purposes.

Chapter 5

Components

5.1 The database

In order to securely store various types of information, a PostgreSQL [3] database has been configured within a `docker` [42] container. This database is storing the personal data of athletes. Moreover, it also serves as a central repository for housing studies, questionnaires, and individual questions. To increase the security of the database, direct access to the database is limited solely to the REST server. Within the database, a total of 35 tables have been designed to cover to the diverse needs of the project. For tables requiring a primary key, a universally unique identifier (UUID1) has been employed as the identifier. These UUID1 keys are automatically generated and updated by the ORM layer.

While the database serves as a unified storage solution for the entire project, it is worth noting that certain tables have been specifically designated for particular modules. This modular approach ensures that the database remains organized and optimized for each component's specific requirements. Indeed, the database formerly had 17 tables but was updated with new tables in order to handle the requirements of the calendar and medical modules. As this thesis worked on the administrator module, it was not impacted by these changes. Moreover, no change in the database structure has to be performed as all the necessary information was already present.

The structure of the database is presented in Figure 5.1. Table 3.1 will describe the tables used for the administrator module as it is the one implemented for this thesis.

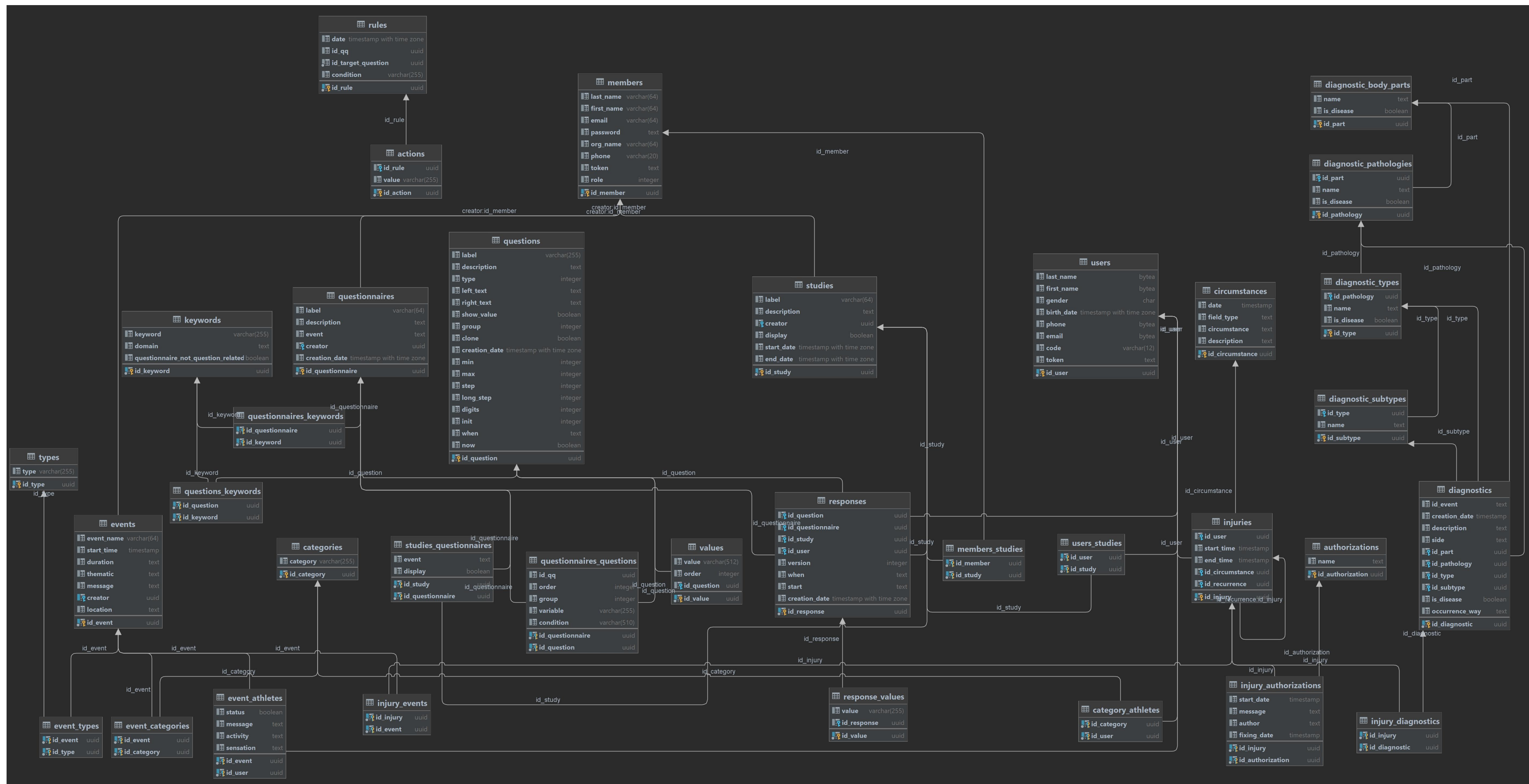


Figure 5.1: Full database architecture

Table Name	Description
members	Represents a member (e.g., trainer/doctor) which can access the administration website. The token field represents a temporary token associated to the member. This token is used to reset the password of a given member. The role field describes the privilege associated with a given member.
users	Represents a user (e.g., athlete) which answers questions using the mobile application. The code field is a 9-digit code used to identify the athlete. In order to comply with GDPR, personal fields (e.g., name, first name, and email address) are encrypted in the database.
studies	Represents a study which is defined by a specific label and description and contains one or several questionnaires. The creator field represents the member who created the study. The display is a Boolean field used by the mobile application to show the study on the homepage. The start date and end date respectively represent the starting and ending dates of the study. These two fields are used by the mobile application.
questionnaires	Represents a questionnaire which is defined by a specific label and description and contains one or several questions. The event field contains a specific temporal representation in JSON which is used by the mobile application.
questions	Represents a question which is defined by a specific label and type and contains one or several values. The type field is an integer that can have different values: (1) text input, (2) MCQ (single response), (3) slider, (4) MCQ (multiple responses), and (5) header/section (delimiter). The left text, right text, show value, and group fields are also used for the mobile application. They represent optional elements that can be used for display.

Table 5.1: Database tables description [41]

Table Name	Description
responses	Represents a response of a specific question, user, and questionnaire. Since there exist recurrent/permanent questionnaires, questions can appear several times. To handle this case, several fields are defined. The start and when fields respectively represent a timespan when the response has been generated and received by the go server. These are also used by the mobile application.
values	Contains the different values (e.g., MCQ, slider entry) of a specific question. Each value is represented as a string.
response_values	Contains the different responses (e.g., MCQ entries) of a specific response. Each value is represented as a string.
keywords	Contains keywords (e.g., tags) related to questions and questionnaires. These keywords are related to particular domains.
rules/actions	Contains rules and actions associated with particular questions and questionnaires. These are used to have conditional questions and questionnaires, allowing for "dynamic" studies.
members_studies	Contains the mapping between members and studies.
users_studies	Contains the mapping between users and studies.
studies_questionnaires	Contains the mapping between studies and questionnaires.
questionnaires_questions	Contains the mapping between questionnaires and questions.
questionnaires_keywords	Contains the mapping between questionnaires and keywords.
questions_keywords	Contains the mapping between questions and keywords.

Table 5.2: Database tables description [41]

5.2 The REST server

5.2.1 Description

The REST server serves as an intermediary between the database and users, trainers, and other members such as administrators or moderators. REST provides a set of conventions and best practices to follow rather than being a standalone technology. This approach allows clients to access web services by exposing specific parts of a program to the external world, commonly referred to as an API.

The REST server, developed in Go, adheres to these conventions and offers various APIs that expose specific interfaces. The following interfaces are provided :

- creation/modification/deletion/listing of studies
- creation/modification/deletion/listing of questionnaires
- creation/modification/deletion/listing of questions
- creation/modification/deletion/listing of athletes (aka users)
- creation/modification/deletion/listing of members (admin, moderators, ...)
- adding answers (of a athletes) to a specific question
- Listing of answers.

5.2.2 Architecture

The REST server follows the Controller-Model pattern. This architecture allows to separate the part of the server interacting with the web module and the one interacting with the database :

- The Controller component focuses on executing high-level procedures when a request is received. These procedures involve tasks like parsing path parameters, decoding JSON, and checking UUIDs. After completing these initial tasks, the controller delegates remaining responsibilities to the corresponding model component. Its role is to coordinate the flow of tasks and interactions, ensuring smooth coordination between the incoming request and the model component.

- The Model component handles data representation and management, including SQL tables defined using Go structures with JSON annotations. It defines the table structure and implements the necessary logic for interacting with them. Interactions with the PostgreSQL database are facilitated through GORM, a database access tool.

With this structure, a operation requested by the user is performed following these steps :

1. From the web app or the mobile app, the user sends its request to the server.
2. The Controller part of the server firsts check if the token are valid. It then decodes the JSON file an parse the different parameters. If no error occurred, it sends the data request to the Model part. This is done internally in the server with a regular function call.
3. The Model uses the parameters to generate to database query using GORM (for some cases raw SQL request might be performed).
4. The Model returns the response to the Controller.
5. The Controller simply transfers this response to the User.

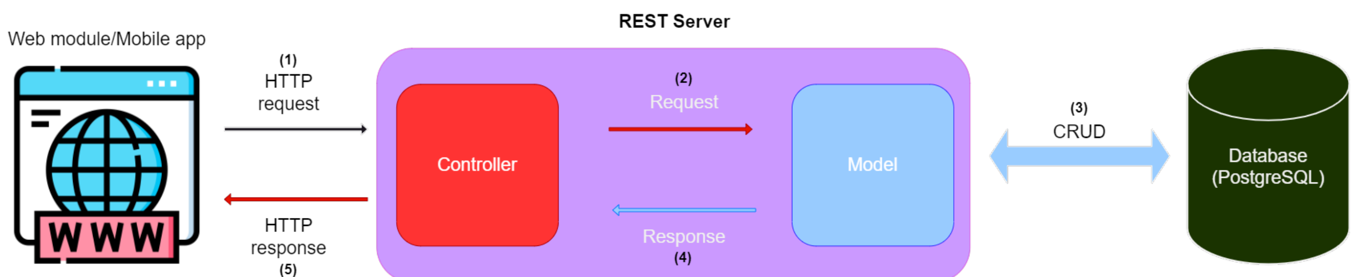


Figure 5.2: REST Server

The server makes use of different third-party dependencies in order to adequately implement the different functions. The most notable ones include :

- Mux : Implements a request router and dispatcher for matching incoming requests to their respective handler [43].
- GORM : Provides an ORM which aims to encapsulate SQL requests and database interactions [6].
- Swaggo : Generates RESTful API documentation with Swagger 2.0 for Go [44].
- JWT-go : Provides a Go implementation of JSON Web Tokens and is used for the authentication [45].

On top of the Controller-Model pattern, a good hierarchy of the packages used must be conducted in order to keep a good modularity. The different packages are organized following the Table 5.3 definition.

Package Name	Description
`auth`	Contains the authentication component of the Go server: manages JWT tokens and user(s)/member(s) sessions.
`controllers`	Contains the high-level procedures (e.g., JSON decoding, UUID checking, etc.) which process a request before dispatching it to the underlying relative model.
`docs`	Automatically generated by Swaggo, it contains the API documentation of the server. (See Section 4.2.7 for further details).
`html`	Contains the HTML code sent by email to a new member after registration.
`models`	Contains the different representations of the SQL tables (as a Go structure with JSON notation) and interacts with the database using GORM.
`res`	Contains sample data that can be used to populate the database (in debug mode).
`routes`	Manages the different routes and roles to interact with the server.
`swaggo`	Defines additional structures that are only used for the API documentation.
`tests`	Contains unit and integration tests.
`utils`	Utility functions and structures (e.g., regex check, etc.).

Table 5.3: Packages used by the server [41]

5.2.3 Authentication

To ensure secure authentication, the server uses JSON Web Tokens (JWT) [45]. Once a user successfully logs in, subsequent requests include the JWT token, granting access to authorized routes, services, and resources. In this architecture, each request incorporates a "bearer authentication" header that contains essential information. The "Id" field corresponds to the UUID of the user or member. It serves as a validation mechanism, verifying the existence of the corresponding identifier in the database. The "Role" field denotes the current user's role, which is verified and assigned by the server. Further details regarding the available role values are provided in the next subsection. Lastly, the "sessionId" represents the ongoing session of the user and is only used with the mobile application. This authentication process ensures the integrity and security of user interactions, providing controlled access to various system functionalities based on their assigned roles and session status.

5.2.4 API

Documentation

For the documentation, the server incorporates the use of swaggo [44], a powerful tool that facilitates the generation of API documentation. By leveraging Go comments with specific and predefined annotations, swaggo enables the automatic creation of detailed API documentation. All relevant comments associated with swaggo are written in the controllers package. Once the server is launched, the API documentation becomes accessible online, which helped understanding quickly some specific points during the back-end development.

```
// GetUser godoc
// @Summary Get a specific user from the database.
// @Description Get a specific user from the database and returns a json which contains the user data.
// @Tags users
// @Accept json
// @Produce json
// @Param id_user path string true "UUIDv4 of the user to retrieve"
// @Success 200 {object} swaggo.JSONResultOK{data=models.User} "Success"
// @Failure 400 {object} swaggo.JSONResultKOBadRequest "Malformed request"
// @Failure 404 {object} swaggo.JSONResultKONotFound "Data is not found (empty)"
// @Failure 420 {object} swaggo.JSONResultKOJWT "JWT token related error"
// @Failure 500 {object} swaggo.JSONResultKOInternal "Internal Server Error"
// @Security BasicAuth
// @Router /users/{id_user} [get]
```

Figure 5.3: Example of Swaggo annotation

Testing

The other that eased the development is Postman [46], a user-friendly application that streamlines the process of making HTTP requests. Postman offers an intuitive and visually appealing interface that eliminates the need for manually coding each request to validate the functionality of the API. A dedicated Postman environment and collection were provided at the beginning of this project to have a better understanding of the API. During the development of this API, the added functions could easily be tested while adding them to the collection in order to make it more complete for further developers.

Roles

To have a separation of responsibilities and access control within the system, the REST server incorporates a role-based authorization mechanism. As the server provides various APIs with different levels of functionality, it needs to differentiate the permissions granted to different types of clients. In order to achieve this, distinct roles have been defined: ADMIN (1), MODERATOR (2), CONSULTING (3), and USER (4). These roles are included in each API request, allowing the server to know the appropriate level of authorization for different clients.

5.3 The Web application

As explained in the section 1.2, *ATHLETin* is composed of several modules, each designed for specific tasks. Each module is implemented in Flutter, the way they are deployed is described in the following Chapter. The administrator module, implemented for this thesis, will be detailed in the next section.

5.4 The mobile application

To answer questions, athletes must install a mobile application developed with the Flutter framework, compatible with iOS and Android. The application communicates with the REST server to retrieve studies, questionnaires, and associated questions. Athletes individually answer the questions and submit them to the server, where the answers are stored in the database.

This mobile application facilitates data collection, allowing athletes to answer the questions any time (e.g. right before and after a training session). By leveraging Flutter's versatility, the application ensures compatibility across different operating systems and makes the maintaining of the whole project easier with only one framework for both the web app and the mobile app.

Chapter 6

Administrator module

6.1 Overview

As one of *ATHLETin*'s main goal is to avoid injuries, collecting feedback from the athletes is a crucial task. Even though the answers are provided using the mobile app, all the treatment of the questions and results must is managed by the administrator module. Indeed, as some data must be kept private, only authorized people will be able to manage such data. Thus, the role of this module will mainly consist of questions and answers management.

6.1.1 Questions management

For this module, a question is defined by :

- its unique ID
- its label : the module should be able to modify it
- its description : the module should be able to modify it
- its type : it can be a MCQ, a slider question or an open question

As thousands of questions will populate the database, a first layer is introduced : the questionnaire. As suggested by its name, a questionnaire is a group of questions. Its properties are :

- its unique ID

- its label : the module should be able to modify it
- its description : the module should be able to modify it
- its group of questions : questions can be added/removed
- its event : this will determine when the questionnaire is active (i.e. the athletes can answers questions with the mobile app)

The questionnaires will then usually be created depending on the time they must be answered. For example, if a group of questions must be answered before and after a training session, 2 different questionnaires will be created even if the questions are the same.

Finally, a study layer is added in order to link questionnaires that must be answered by the same group of athletes. A study is composed by :

- its unique ID
- its label : the module should be able to modify it
- its description : the module should be able to modify it
- its group of questionnaires : questionnaires can be added/removed
- its group of athletes : athletes can be added/removed
- its start and end date : the time zone in which the study is active

Along with their main features, questions, questionnaire and studies are also characterized by other fields such as their creation date or their creator.

6.1.2 Results management

As explained before, the same question can appear multiple times inside a study. In order to distinguish the answers, a response is defined using :

- its unique ID
- the ID of the corresponding question
- the ID of the corresponding questionnaire

- the ID of the corresponding study
- its value
- the time of the response (i.e the moment the athlete answered the question)

With all these pieces of information, the module must be able to select responses depending on provided criteria (getting all the responses from a user or from a study) and export them as a `csv` file.

6.2 Pages

In order to handle efficiently the different parts of this module, a correct page separation must be determined.

6.2.1 Login Page

It is the first displayed page when launching the module. As it is reserved for administrators, a login form is obviously required. Once connected, the JWT of the user will be stored to be added in each request.

6.2.2 Front Page

This simple page lists all the other pages of the module. The user might be redirected to this page in case of error this will depend on the type of error). It also features a button to logout.

6.2.3 Try Later Page

This page will be displayed only in case of error. More precisely, it will happen when a request receives an error code of 500 because the server is not available.

6.2.4 Studies Page

All the studies are listed on this page. A search bar is included to easily find a study if needed. From this page, two other pages are accessible. These two sub-pages are specific to each study.

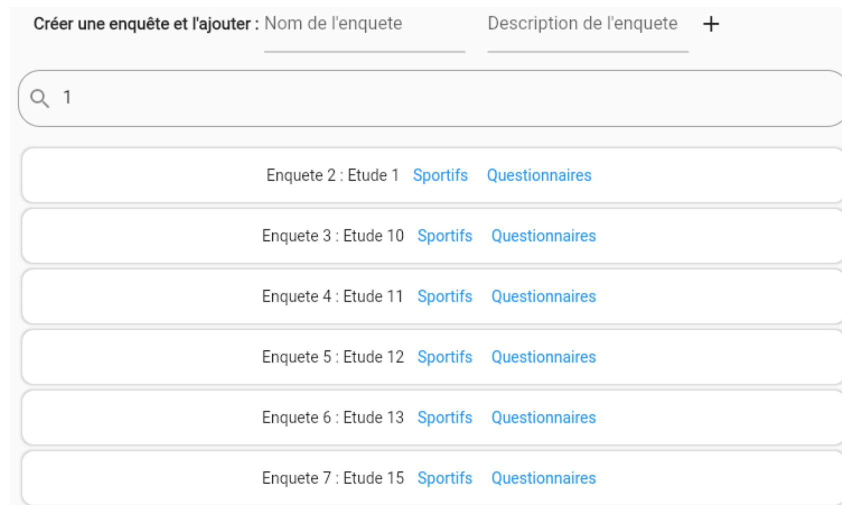


Figure 6.1: Studies list filtered using search

Study/Athletes Page

This page lists all the athletes in a table their first name, last name and ID. It also indicates if they are registered to the study with a button allowing to register/unregister them.

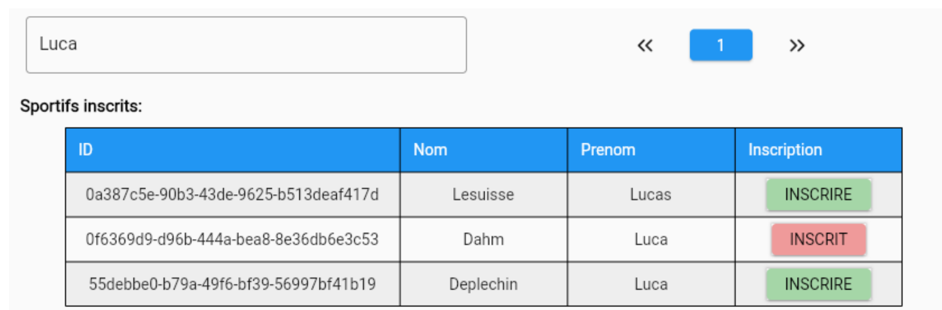


Figure 6.2: Athletes list filtered using search

Study/Questionnaire page

The questionnaires belonging to the study are displayed here. They are structured as a `ListView` of `ExpansionTile`, meaning that only their label is displayed before expansion. A drawer is included to facilitate the experience by allowing to search for a specific questionnaire. When selected, the drawer will scroll to the questionnaire

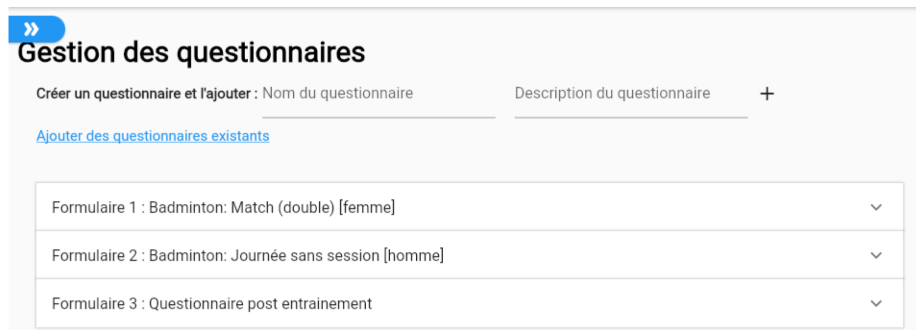


Figure 6.3: List of questionnaire from a study

and expand its tile. On top of the page is the creation form. By simply providing the label and the (optional) description of a questionnaire, a new one is generated and added to the study. The button to add existing questionnaires opens a dialog that will list all the questionnaires in a table similar than the one listing the athletes.

When expanded, a questionnaire provides the list of its question. This list is also a `ListView` of `ExpansionTile`. Currently, the question tile only displays their description when expanded but more features could be proposed in future works. There a four buttons allowing to update the questionnaire :

- Clone questionnaires : it allows to add a group of question to the questionnaire by directly copying the ones of another questionnaire. The selection of the questionnaire to clone is similar to the selection of questionnaire to add to the study.
- Add questions : will display the list of questions in a table with a button to add/remove each question to/from the questionnaire.
- Update questionnaire : allows to change the label or the description of the questionnaire.
- Events : configure the event of the questionnaire. More details will be provided in the following subsection.

6.2.5 Events Page

Some questionnaires require to be fulfilled only at specific moments. To implement that, the module generates events. There are multiple ways to chose the time of activity of a questionnaire. This leads to different types of event.

Daily Fixed Event

This is the simplest form of event. It is defined by :

- Time Periods : This corresponds to the start and end time of the event. There can be multiple time periods for Daily Fixed Events. The number of time periods will then correspond to the number of occurrence of the event and the length of a time period will correspond to the duration of the event.

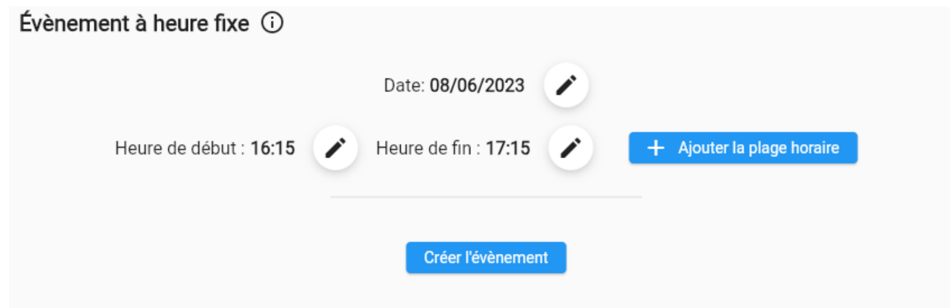


Figure 6.4: Daily fixed event form

Daily Random Event

This type of event occurs at random moments depending on :

- Time Periods : These are the periods during which an event is susceptible to happen. They must be chosen large enough to fit with the other parameters. Indeed, if a period is too short, it would be impossible to assign it an event.
- Duration : This is the length of an event. If there are multiple occurrences, all the event will have the same length.
- Occurrences : The number of time an event happens.
- Guard : If there are multiple occurrences, this is the minimum duration between the end of an event and the beginning of the next one.

Figure 6.5: Daily random event form

Daily Event

Unlike the two previous types of event, Daily Events can be tuned to occur on specific days. This tuning relies on :

- Weekdays : From Monday to Sunday, the user checks the days where he wants the event to be active.
- Event : The events happening on selected days can be either Daily Fixed Events or Daily Random Events.
- Start date : The beginning date of the event.
- Delta 0 : This corresponds to the offset (number of days) added to the start date to fix the first event.
- Delta F : This offset corresponds to the maximum number of days after the Start Date where the events can be active.
- not Until : This value is a weekday, it specifies the day from which the events can start.

Figure 6.6: Daily event form

6.2.6 Results Page

This page is responsible for results fetching. As previously explained, the questions are parts of questionnaires which are themselves grouped into studies. The result can be obtained for a specific questionnaire of a study. The user simply needs to provide the id of the study and the one of the questionnaire. Two options are then available :

- Show the results : this will display the results of the questions in a table with the row corresponding to the users and the column representing the questions of the questionnaire.
- Download the results : The table of results having the same structure as the shown one will be downloaded under the `csv` format. The user will then be able to perform specific queries with a dedicated software handling `csv` files.

To allow some flexibility in the results fetching, a filter with advanced parameters is provided. This filter will be applied on the **athletes** and has the following properties :

- Users id : will fetch the results only for questions answered by athletes who's id is provided.
- Gender : will fetch the results for men/women.

- Birth date min/max : will fetch the results only for athletes born after the min date/before the max date.
- Studies id : will fetch the results for athletes registered to the study, it won't fetch the results of the study except if the same id is provided in the main study field.
- Questionnaire id : will fetch the results only if the athletes answered the provided questionnaire. For example getting the results for a questionnaire happening after a training session only if the athletes answered the questionnaire before the training session.
- Question id : will fetch the results only for athletes who answered the provided question.
- Response value : will fetch value only if the athletes gave a specific answer. For example, getting the results for athletes having a pain in the leg.
- Response date min/max : will fetch the results only if the athletes answered after/before a specific date.

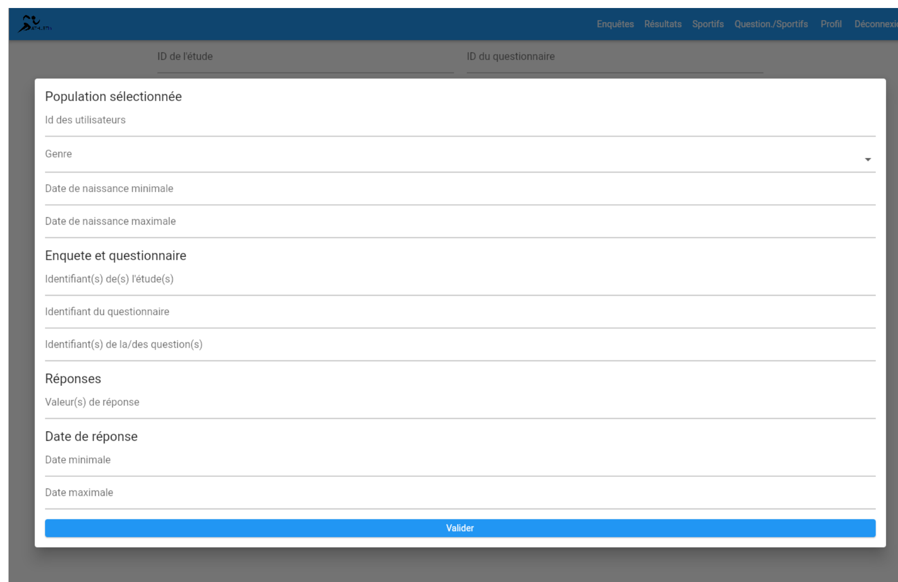


Figure 6.7: Filter of the result page

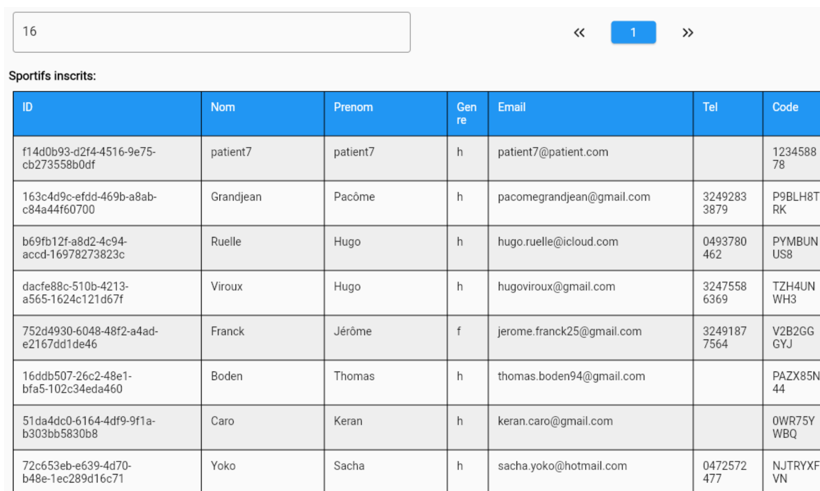
6.2.7 Athletes Page

The list of all the registered athletes is provided in this page. They are displayed in a table listing their :

- Id
- **Last name**
- **First name**
- **Gender**
- **Email**
- **Phone Number**
- Code

The page features the form to create a new athlete by providing its **personal information**.

As the app manages a high number of athletes, pagination is used to lower the loading time. A search bar is also available to quickly find an athlete from its id, last name or first name. The implementation details on these mechanism will be introduced in the following sections.



ID	Nom	Prenom	Genre	Email	Tel	Code
f14d0b93-d2f4-4516-9e75-cb273558b0df	patient7	patient7	h	patient7@patient.com		123458878
163c4d9c-efdd-469b-a8ab-c84a44f60700	Grandjean	Pacôme	h	pacomegrandjean@gmail.com	32492833879	P9BLH8TRK
b69fb12f-a8d2-4c94-accd-16978273823c	Ruelle	Hugo	h	hugo.ruelle@icloud.com	0493780462	PYMBUNUS8
dacfe88c-510b-4213-a565-1624c121d67f	Viroux	Hugo	h	hugoviroux@gmail.com	32475586369	TZH4UNWH3
752d4930-6048-48f2-a4ad-e2167dd1de46	Franck	Jérôme	f	jerome.franck25@gmail.com	32491877564	V2B2GGGYJ
16ddb507-26c2-48e1-bfa5-102c34eda460	Boden	Thomas	h	thomas.boden94@gmail.com		PAZX85N44
51da4dc0-6164-4df9-9f1a-b303bb5830b8	Caro	Keran	h	keran.caro@gmail.com		0WR75YWBQ
72c653eb-e639-4d70-b48e-1ec289d16c71	Yoko	Sacha	h	sacha.yoko@hotmail.com	0472572477	NJTRYXFVN

Figure 6.8: List of athletes filtered (by ID) with searching

6.2.8 Athletes/Study Page

While the study page allowed to register multiple users to a study easily, this page does the opposite by allowing to register one user to multiple studies. All the users are listed in `ListView` of `ExpansionTile`. On expansion, the list of study is displayed in a table where the athletes can be registered.

A drawer is available to search for an athlete. When selected, it will automatically scroll to the athlete's tile and expand it.

1 : patient1 patient1(321ee98d-6da3-4f97-ae6b-bcdcff0be2d3)

Inscription aux enquetes:

#	Enquete(s)	Date debut/fin
1	Enquete BO	INSCRIRE
2	Etude 1	INSCRIT
3	Etude 10	INSCRIRE
4	Etude 11	INSCRIRE
5	Etude 12	INSCRIRE
6	Etude 13	INSCRIRE
7	Etude 15	INSCRIT
8	Etude 2	INSCRIRE
9	Etude 3	INSCRIRE
10	Etude 4	INSCRIRE
11	Etude 5	INSCRIRE
12	Etude 6	INSCRIRE
13	Etude 7	INSCRIRE
14	Etude 8	INSCRIRE
15	Etude 9	INSCRIRE
16	Nouvelle enquete	INSCRIRE

2 : patient2 patient2(cd6c2c2e-9953-492d-b2e1-038ba58ac241)

Figure 6.9: Expanded Tile to register an athlete

6.2.9 Members Page

Similarly to the users page, this page lists the members. The form allows to create new members and to attribute them a role between :

- Administrator
- Moderator

- Consultant

6.2.10 Profile Page

This page simply shows the information about the user such as its last name, first name, role,...

6.2.11 Page routing

Figure 6.10 displays the routing of the module. This routing was implemented using Flutter's named route. On top of this, every page is able to navigate to `FrontPage` and its six sub pages using the app bar.

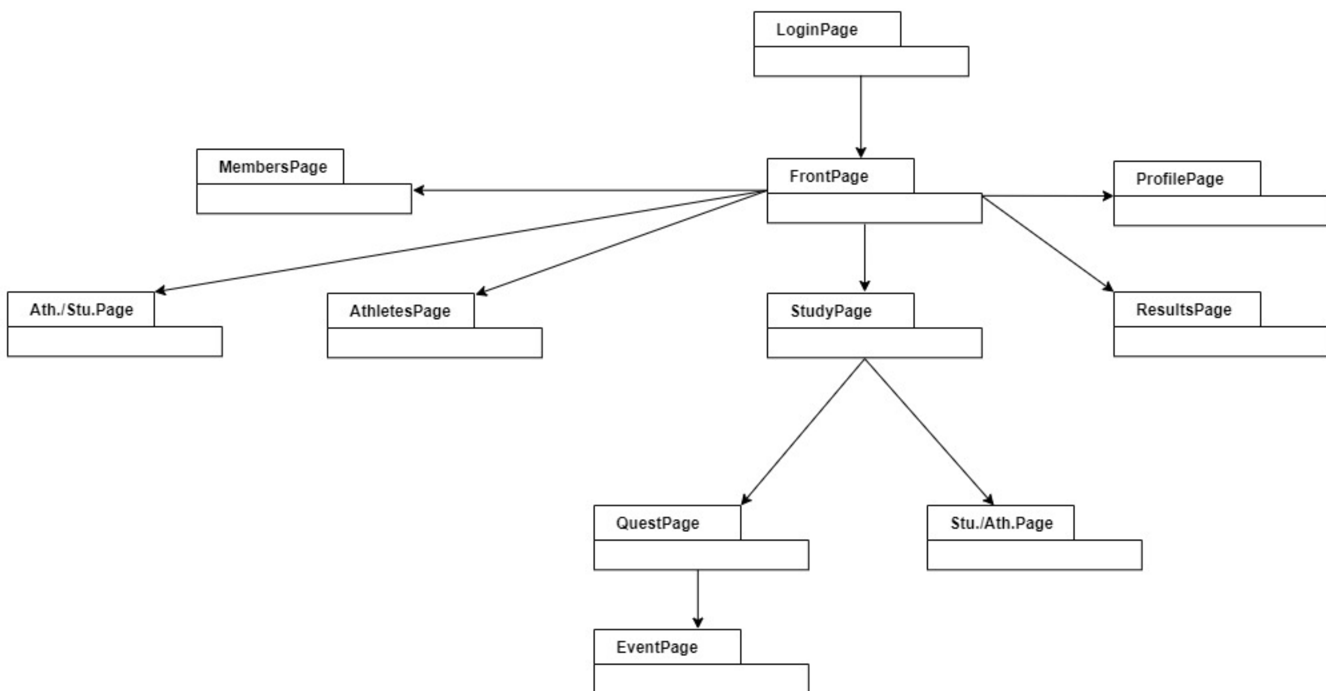


Figure 6.10: Page routing in the administrator module

Chapter 7

Implementation structure

The different pages were designed following specific patterns for the code design. The widgets used necessitated specific structure in order to correctly fulfill their tasks. The following chapter will go over the architecture of the code and some technical details worth mentioning.

7.1 Architectural Pattern

The module mainly operates by fetching data from the database and displaying them. It also allows to update or create some content. To make these operations efficient and well organized in order to handle further modifications, a good design pattern is necessary.

7.1.1 Model View Controller pattern

The Model View Controller pattern [47] aims to distinguish the logic of the application from its interface. To do so, 3 components are defined, each having their specific roles and interactions :

- The Model : It represents the application's data. It will also integrate the business logic to ensure data integrity. When a change occurs in the data, it must notify the View.
- The View : The user interface is under the responsibility of the View. It obtains the data to display from the model and will relay user's inputs to the Controller.

- The Controller : It is the intermediary point between the View and the Model. It will handle the inputs provided by the View in order to adequately update the Model.

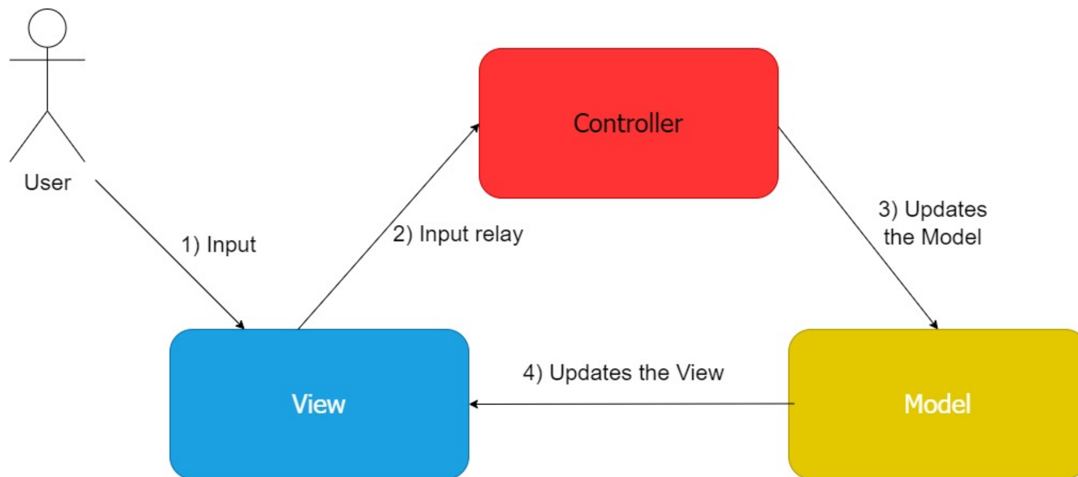


Figure 7.1: Model View Controller pattern

For example, the athletes page of the administrator module would be structured as follows :

- `User.dart` : The class User (which represents athletes) is defined with the different methods involving athletes (get the list of athletes, create a new athlete,...)
- `athletes_page.dart` : This is the implementation of the page described in the previous chapter. This list of athletes is requested from the file User.dart and displayed in the table. Moreover, the athlete creation form will call the controller when an athlete is created.
- `athletes_page_controller.dart` : The methods are called by the user when he interacts with the page. When called, the will call the methods of `User.dart` to update the model.

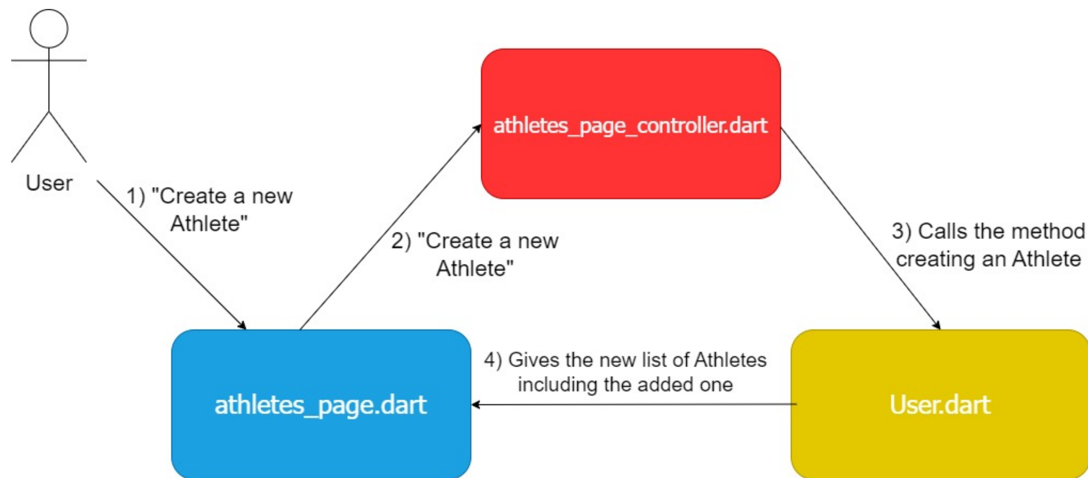


Figure 7.2: MVC example applied to the module

This pattern was initially chosen for its simplicity and to keep the same global design as the server which used a Controller-Model pattern. However, some limitations were met during the development. This lead to a change for the Model View View-Model pattern that will be described in the next subsection.

7.1.2 Model View View-Model pattern

During the development of the module, the user interactions lead to complex logic in order to correctly update the view. Indeed, the addition of features like searching or pagination lead to multiple operations to correctly handle the inputs and update the view. All these operation would lead to an overloaded Model component while the Controller would mainly be calling the Model's methods. As the Models are defined by `Dart` classes, only the fields and basic methods would be implemented there. The complex logic would then be transferred to the Controller as it has access to all the data from the View and the Model. This new pattern choice actually corresponds to the Model View View-Model pattern [48] [49].

This pattern is defined by :

- The Model : Like in MVC, the model represents the data of the app and check their integrity. However, the change in the data will notify the View-Model (and not the View like in MVC).
- The View : UI is still managed by the View. The only connected component is the View-Model which will update the View if necessary.

- The View-Model : Placed between the View and the Model, it will play an important role in communication. It must modify the View depending on the changes in the Model.

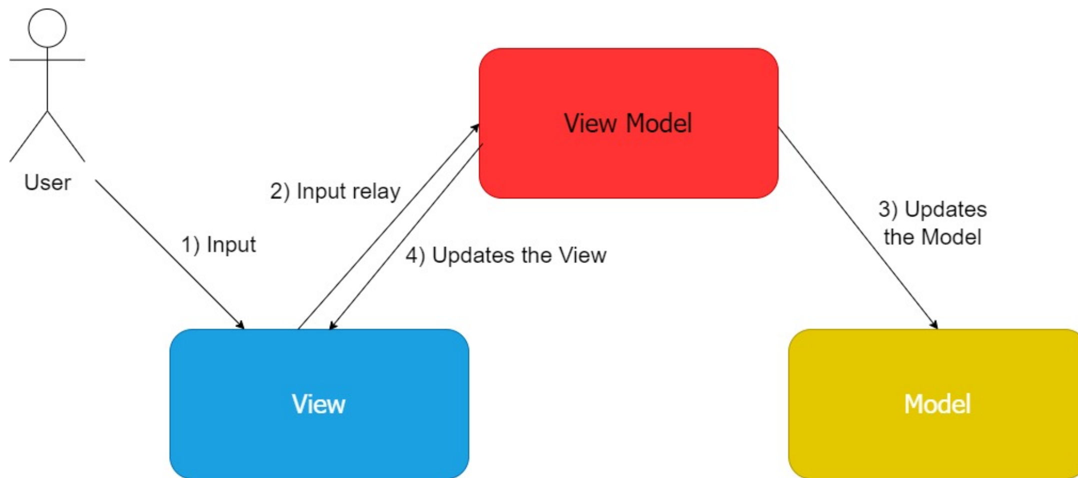


Figure 7.3: Model View View-Model pattern

This pattern lead to the following roles for the implemented files (who kept their initial names) :

- `User.dart` : This `Dart` class still defines an athletes with its fields and methods.
- `athletes_page.dart` : This page is now only linked to `athletes_page_controller.dart`. It will get the information on the list of athletes using a `watch` method of the package `Provider` to keep track of the variables stored by `athletes_page_controller.dart`. When receiving an input, it calls the appropriate method of the View-Model and waits for updates.
- `athletes_page_controller.dart` : All the variables used to set up the view are stored here. Some will be watched by the View while other are only internally used in order to update the watched variables. After its initialization, it will wait for input relay from the View and perform the required operation to update it.

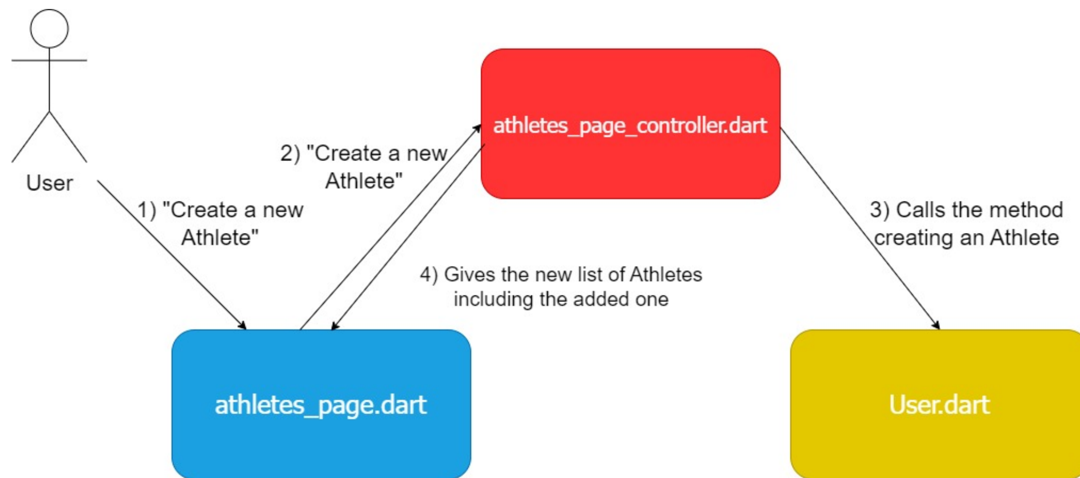


Figure 7.4: MVVM example applied to the module

7.2 Services

When developing a web module, some functions are regularly called. In order to simplify the structure of the code, these recurrent functions are grouped into services.

7.2.1 Navigation service

The management of the `GlobalKey` of the navigator state is handled by this service. It will the allow the page to call simpler functions to navigate without worrying about the `GlobalKey`.

7.2.2 Identity service

The information about the account logged in are stored by this service. Mainly used by the `Login Page` and the `Profile Page`, this service is also required every time a request is made by the `Network Service`.

7.2.3 Network service

All the HTTP request are performed via this service. It performs :

1. Writing of the complete URL
2. Writing of the header including the token authentication process

3. Decoding the response from the server
4. Catching exceptions to handle request related errors

7.2.4 Provider Service

To avoid declaring many `providers` in the `main.dart` file (using the `MultiProvider` class), only one general `ProviderVariables` is declared (along with the Identity Provider). All the controllers are referenced by the `ProviderVariables` that will create an instance or reference to its existing controller if it already exists.

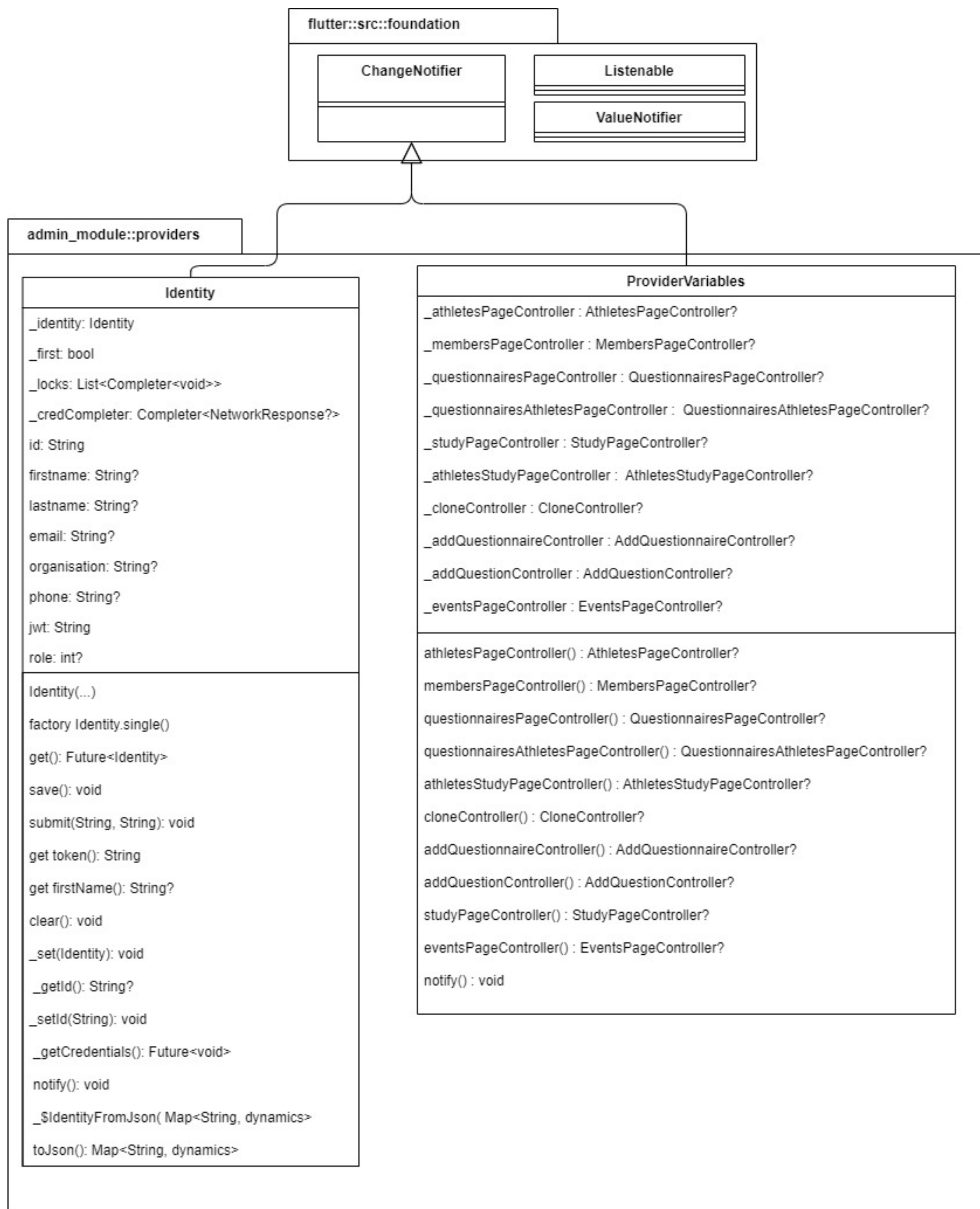


Figure 7.5: Provider and Controller structure for the services

7.3 Files organization

At the beginning of this thesis, a template was provided in order to set a structure for the project. The development was performed in adequacy with the initial structure while adding some folders when necessary. Overall, the MVVM pattern defines the main packages composing the project. Eventually, the module is divided into several packages presented in Table 7.1.

Package Name	Description
`assets`	Stores the different images used by the app.
`components`	Stores the widgets used recurrently such as the `AppBar`.
`controllers`	Stores the View-Models of the pages.
`models`	Stores the Models of the pages.
`pages`	Stores the Views of the pages.
`services`	Stores the different services implemented.
`theme`	Stores the constants such as the address and port used in the requests.
`tests`	Stores the unit tests.
`utils`	Stores utility functions.

Table 7.1: Packages used by the module

7.4 Performance

When developing a module, performance should play a fundamental role in the various choices made throughout the project. Indeed, the app needs to be fluid and minimize as much a possible the different loading times. To do so, several features were integrated in the app.

7.4.1 Caching

One of the most common practices to improve the performance of a website is the use of caching. This thesis was no exception as site caching was implemented in order to avoid repetitive requests when querying for content used at different places of a page.

Caching is performed as page scale, meaning that the cached data's lifetime ends when the user visits another page. The storing being managed by the controllers of each page, they are reset when a page is visited again.

To reduce the number of interaction, the pages controller directly update the cached data when the user wants to modify data. To ensure data integrity, the modification of cached data occurs only when the server sends a message informing the true data were successfully updated in the database.

7.4.2 Pagination

The fluidity of the app is a crucial point. To minimize the loading time of the pages, some pages uses pagination. At initialisation, only the data listed in the first page are requested to the server to have a lighter response and less initialization time. When the users visits other pages, the request is made to the server for the corresponding page number. The pagination is also using the caching. Indeed, the controllers store a `Map` associating each page number to the data corresponding to the page. With this, the page content will be requests only once to the server and will then be directly available when the user comes back to a previous page.

To handle the pagination, the View needs 3 variables. These are obtained and updated using the `watch()` method of `Provider`. For example, in the `Athletes Page`, the file `athletes_pages.dart` tracks the values of :

- `userList` : This is the list of users displayed in the table. Its length will be lower or equal to the page size (which is fixed for all the app).
- `userCount` : This is the total number of Users, including the one not displayed. This variable is used to know the total number of pages to allow the user to go directly to the last page if needed.
- `userPage` : This is the current page number. It's initialized at 1 and will be updated depending on the user's actions.

When the user changes the page number, the input will be handled in 3 different ways depending on the context :

1. Search is active : If the user is using the search bar, the controller won't need to request the server for the new page. Indeed, the controller will already have a list of filtered users (more details in the following subsection) so it only needs to modify the range displayed by updating the variable `userList`.
2. The page was already requested : As explained before, the controller uses caching so previously requested variables are still stored. No request will be

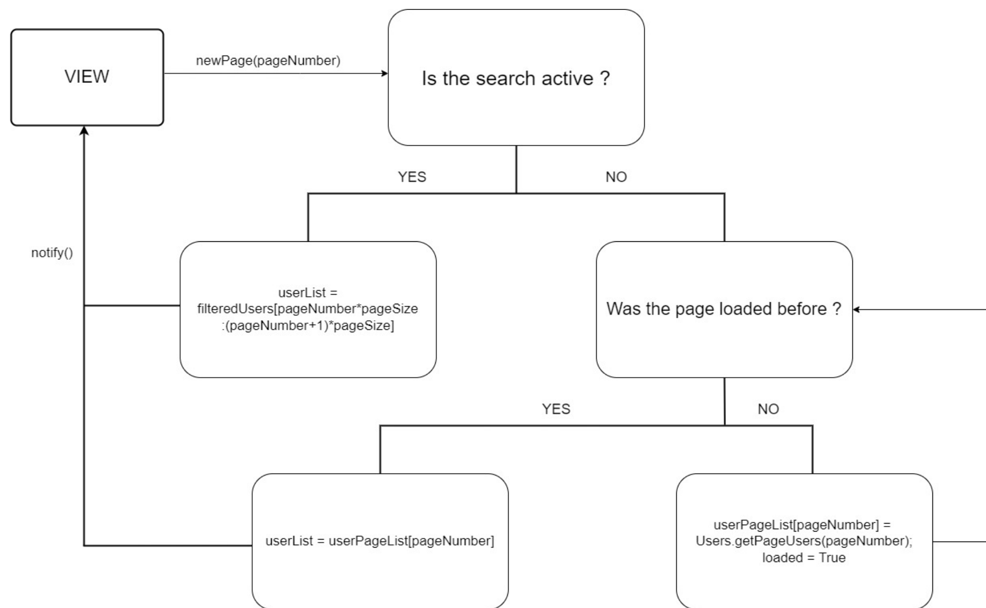


Figure 7.6: Pagination mechanism applied to the `Athletes Page`

performed and the `userList` variable will be update by the list stored at the index `pageNumber` of the map `userPageList`.

3. First time the page is loaded : The controller request the list of Users corresponding to the page number then stores it in the `userPageList`. It then performs the same steps as in case 2.

The page size was set to 20 for this module. It can be updated depending on the priority in the trade-off between the number of interactions with the server and the initialization time.

7.4.3 Searching

Unlike caching and pagination, searching functionality is not a feature that will directly make the app faster by reducing loading time. The true benefit of such tool is to enhance the user experience by allowing to directly access a specific resource. Eventually, it will lead to less loading time and server requests as the user wouldn't need, for example, to go through all the pages to find the athletes he is looking for.

In the module, two different implementations of searching were used :

- Intern searching : All the data are already present on the page, it will internally filter the displayed data depending on the search input.
- Server searching : The data is not yet fetched from the server. A request will be sent by specifying the search input in order to directly receive the data corresponding to the search criteria.

As the server searching will require to perform requests, the search input need to be correctly handled to avoid sending too many requests in a short amount of time. To do so, a cooldown timer of 500 milliseconds is set after each search request.

On the server side, the search is done in 2 steps :

1. Fetch all the elements from the database
2. Use Go routines to filter all the data in parallel

The search couldn't be operated directly on the database (using the `LIKE` operator) because some values need to first be deciphered.

Once a search result is received by the module, the page number is set back to 1 and the total number of pages is updated depending on the length of the received list. When the input search is fully erased, the page 1 is requested again from the server to refresh the cache.

7.4.4 Measuring the performance

Flutter DevTools

Dart and Flutter provide tools allowing to get better insights on the way the app is run [50]. This can enlight problems in the code such as unnecessary operations that would lead to bad performances.
The tools available for the web application include :

- The Flutter Inspector : It gives the details about the widget tree and the different rebuild by highlighting the widgets in the app [51].
- The Debugger : Acts as a regular debugger, allows to stop the app on breakpoints, to analyze some variables, to execute step by step,... [52]

Other features were implemented by Flutter such as a CPU and Network profiler but they were not available for this module.

Browser tools

To have more insights about the performance of the app, the browser can be used to directly analyze the behaviour of the module. `Chrome DevTools` [53] were used when analyzing the app.

They allow :

- CPU profiling
- Network profiling
- Frame rate analysis

To have a general rating of the website, Lighthouse [54] was used. This open-source tool was launched by Google to help developers when they optimize their website.

It gives a score between 1 and 100 to several aspect of the website such as [55]:

- The performance : It makes several time measures to establish a score on the general performance of a page. It was not able to score the module but the performance should be handled internally by Flutter during compilation to JavaScript code.
- The accessibility : A high score means people using assistive technologies can still use properly the website.
- The best practices : Checks if the modern standards of web development are met.
- The SEO (Search engine optimization) score : It checks if the website if optimized for search engines. This will generally mean that the user experience is optimized too.

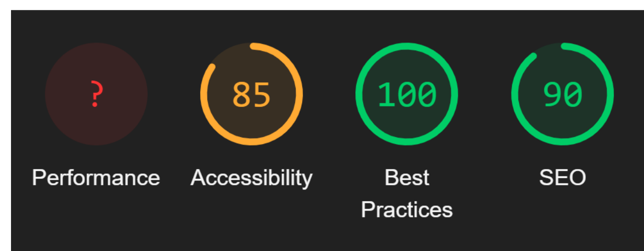


Figure 7.7: Lighthouse score for the `Study Page`

The different tools were mainly used to check if no aberrant value was measured when the app runs. They should mainly be used when a problem is met or when the website runs too slowly. Moreover, these tools are more suited to give good feedbacks to website developed using Javascript. In the context of this thesis, the module is developed in Flutter, which already performs optimization when compiling the Dart code. As the website ran normally (no jank, no loading excessively long), no advanced use of the different profiling tools was needed during development.

Chapter 8

Testing

Testing is an essential aspect of software development as it helps ensure the quality and reliability of applications. This chapter will go over the different types of test that be applied to a Flutter app [56].

8.1 Unit Tests

Unit tests are focused on testing individual units of code, such as functions, classes, or methods, in isolation. They help checking that each unit of their app behaves as expected. In Flutter, unit tests are written using the `Flutter test` framework and can be executed using the flutter test command.

Unit tests in Flutter are useful for testing business logic, algorithms, and data processing operations. Indeed, as they are applied on small part of the code, they are easy to implement and run quickly.

8.2 Widget Tests

Widget tests in Flutter are designed to test the behavior and appearance of individual widgets in an app. Unlike unit tests, widget tests focus on testing the interaction between widgets and their surrounding environment, including rendering, user interaction, and state management. These tests ensure that widgets are rendering correctly and responding appropriately to user input.

As the components tested are less isolated then the ones tested by units tests, the widget tests take more time to run.

8.3 Integration Tests

Integration tests are the most complete type of automated test. They focus on testing the interaction and integration between different parts of an app. These tests validate the behavior of multiple components working together, including the UI, business logic, and external dependencies. Integration tests help ensure that all the components of an app are functioning harmoniously and producing the desired results.

Integration tests are beneficial for catching issues that may arise due to the interaction between different components of an app. They help identify bugs related to data flow, API integration, navigation, and overall app behavior. However, due to their complexity, they take a lot of time to run.

8.4 Manual Tests

For this thesis, a full set of automated set was unfortunately not implemented. During development, the accent was put on the manual test which allowed quicker check at the cost of less cases coverage. As the app contained many recurrent code structure (e.g. tables displaying user, questions, questionnaire), some manual test could be conducted once and validate multiple parts of the module.

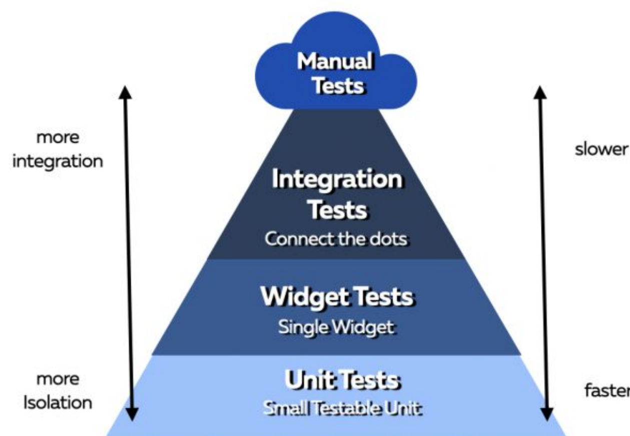


Figure 8.1: Test Types hierarchy [57]

Chapter 9

Deployment

Once finishing the development of the app comes the deployment part. This chapter will go over the structure of *ATHLETin*'s module. It will also explain the mechanism used to group all the modules to simplify the user experience.

9.1 Modules encapsulating

The different modules of the *ATHLETin* project (e.g. the medical and calendar modules) were all encapsulated in containers managing the different dependencies. All the module are encapsulated in different modules to be able to manage their dependencies independently. `Docker` was used to generate the containers.

`Docker` [42] is a popular containerization platform that allows developers to encapsulate their applications and their dependencies into isolated containers. For the app, it makes the containerization simple. Indeed, once the different dependencies are correctly listed in the `Dockerfile`, the the `Docker` image can be directly built. This image can then be run on any environment as long as `Docker` is installed on it.

The image generation was performed in 2 steps :

1. Environment set up : Using a `Debian` [58] image, it installs the dependencies necessary to run Flutter.

2. Run-time image creation : The built web application is copied into a `Nginx` [59] container.

9.2 Reverse Proxy role

As *ATHLETin* is composed of several modules isolated from each other, they must all be accessible within a single web server. This server was developed using `Nginx`. It has been configured as a reverse proxy that will redirect the user to the corresponding module depending on their request. The modules as distinguished from each other by their port number, they all have the same domain.

Part IV

Conclusion

Chapter 10

Conclusion

This thesis described how was designed the administrator module of *ATHLETin*. As this work is part of the complete *ATHLETin* project developed by the professor Laurent Mathy and his team, a general overview was conducted to clarify the context of this work.

The *ATHLETin* module offers several key features, including the Admin Module for managing user accounts and permissions, the Training Module for creating and organizing training sessions, the Agenda Module for scheduling and tracking events, the Medical Module for managing medical information, and the Communication Module for facilitating communication among users. These functionalities provide a comprehensive solution for managing training activities efficiently.

A first description of *ATHLETin* and its goals allowed to understand to main objective of the project. The *ATHLETin* modules offer several key features, including the Admin Module for managing user accounts and permissions. These functionalities provide a good solution for managing training activities efficiently.

To set a common knowledge basis on the tools used for the implementation, a description of `Postgresql`, `Go`, `GORM` and `REST` was provided for the backend part. The `Dart` language and the `Flutter` framework were presented for the frontend part. For every technology, the different reasons leading to their choices were listed. The main one being the scalability and the flexibility to allow further improvements.

From these technologies, the implementation structure of the module followed architectural patterns such as the Controller-Model for the backend and the Model-

View-ViewModel (MVVM) for the frontend. These patterns helped in organizing the codebase and separating concerns, leading to a maintainable and extensible architecture.

With the structure correctly presented, the core of the work was presented by explaining the different pages of the module. The important features were also described along with the way they were implemented.

Once the development part was over, the testing and the deployment were the next steps of the project. Some generalities were introduced as well as the specific case of the project. The way the admin module will be linked with other module was also mentioned.

In conclusion, the development of the ATHLETin Flutter module managed to met the different challenges. The implementation structure, testing , and deployment strategies employed have contributed to the overall quality and usability of the module. As the module can always be improved, room was left for future development adding even more features by using flexible and scalable code.

Chapter 11

Future work

This final chapter will list several improvements or features that would enhance the user experience when using the modules. First ideas will be about the general *ATHLETin* project then specific improvements for the administrator module will be presented.

11.1 *ATHLETin*

11.1.1 Session persistence

As previously explained, the different modules are implemented separately and isolated in containers. When navigating between the different modules, the session should be maintained active to avoid having to reconnect every time. This feature will involve cookies management.

11.1.2 More languages

The different modules have a french interface. Adding different language could not only enlarge the targeted audience of the app, but also help non french speaking athletes already using the app when they answer questions.

11.1.3 Role Management

The administrator module currently allows to create members and attribute them a role (Admin, Consultant, Moderator). However, the different authorization implied by these roles are not completely defined. Moreover, 3 roles might be not enough

to handle the different authorizations due the number of possible operations in the module. A finer role tuning could help the management of athletes throughout the different modules

11.2 Admin Module

11.2.1 Question creation

Studies and questionnaires can be created but the question creation was not implemented. A simple menu could have been sufficient but as question will be one of the components of the database that will take a lot of memory, efficiency in the creation should be taken in account. By efficiency, the goal would be to avoid creating too many similar questions that would lead to useless duplicates in the database. Current features like keywords could be exploited to first search if a question already exists before creating one.

11.2.2 Event Visualization

The different questionnaires are active depending on the time period of their event. Adding a visualization feature would help the management by directly allow the user to know how many questionnaires are active at a specific time period. This can avoid setting too many questionnaire at the same time which would overload the athletes with too many questions to answer.

11.2.3 Improve Performances

The different features set up to have a good performance could be improved with better algorithms. For example, the search method could automatically adapt from external to internal searching after a fixed number of possible choices. The caching mechanism might also be implemented differently, by adding a fixed life time instead of waiting for the page to refresh.

11.2.4 Adding features

The description of the module provided in 1.2.1 lists many features. Some were implemented in this thesis like the data exportation but many are still to be designed.

Appendix A

Module screenshots

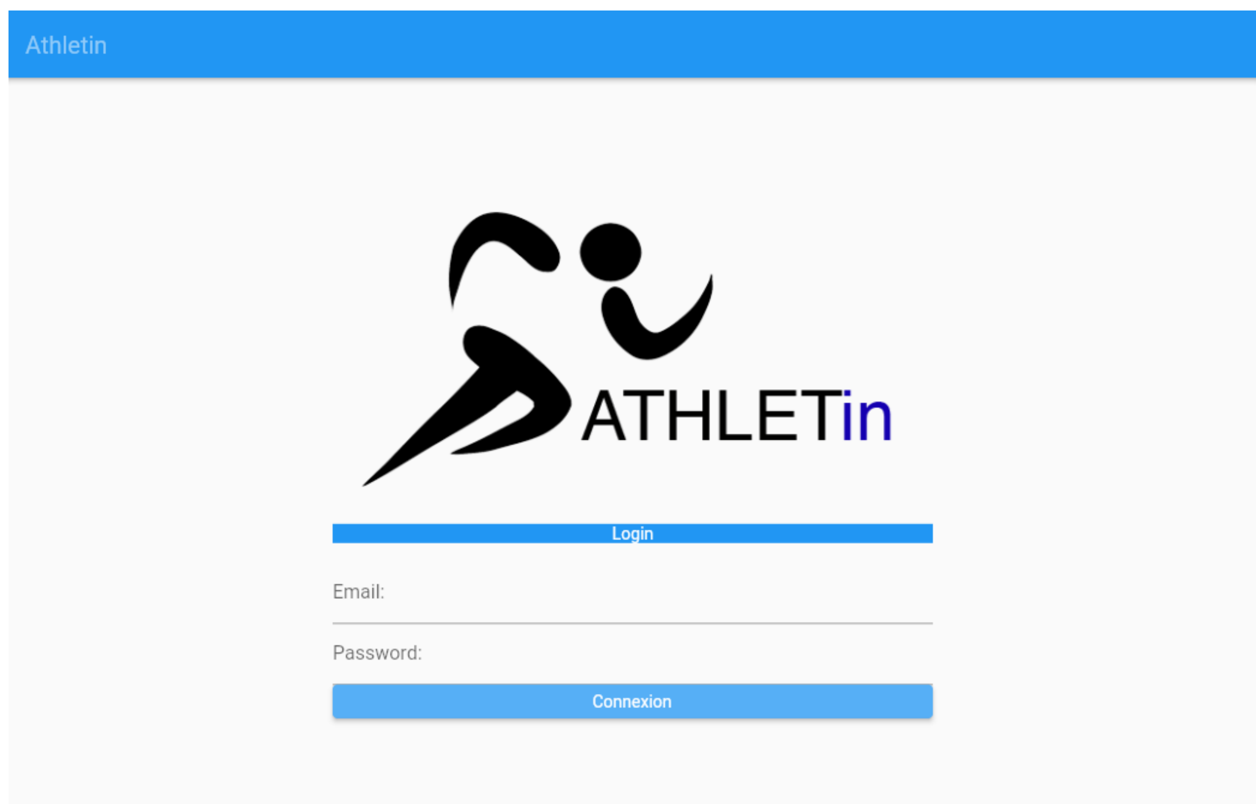


Figure A.1: Login Page

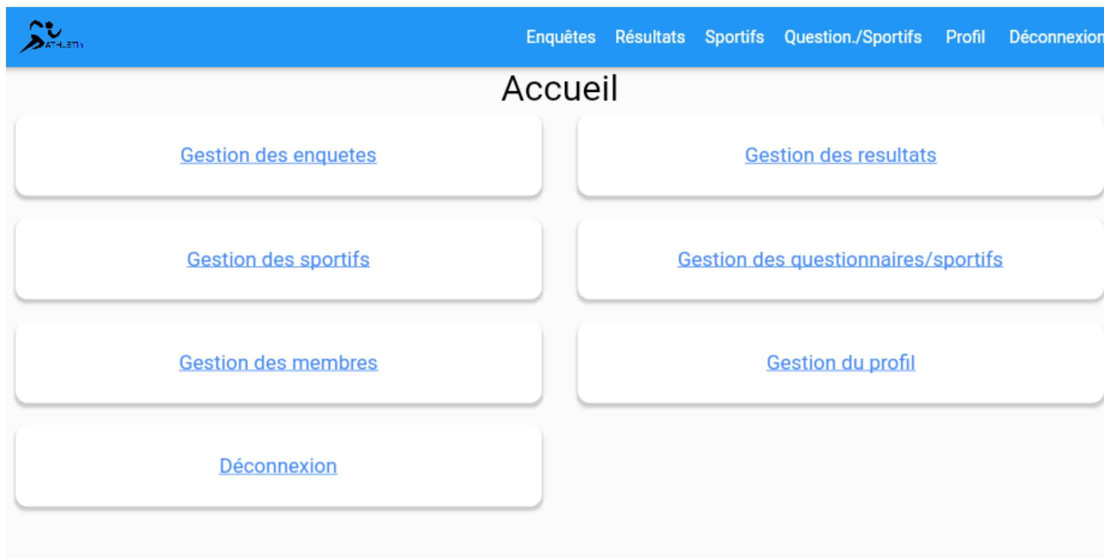


Figure A.2: Homepage

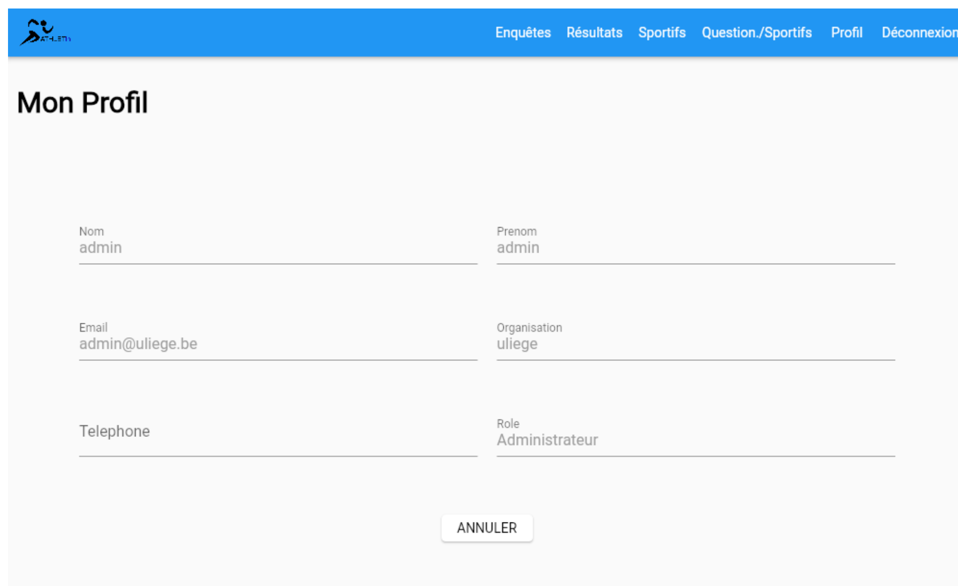


Figure A.3: Profile Page

Enquêtes Résultats Sportifs Question./Sportifs Profil Déconnexion

Gestion des sportifs

Ajouter un nouveau sportif

Nom

Prenom

Email

Telephone

Code
4WKECPNHV

genre

ANNULER ENVOYER

Figure A.4: Form to add an athlete

Rechercher

« 1 2 »

Sportifs inscrits:

ID	Nom	Prenom	Genre	Email	Tel	Code
321ee98d-6da3-4f97-ae6b-bcdcff0be2d3	patient1	patient1	h	patient1@patient.com		012345678
cd6c2c2e-9953-492d-b2e1-038ba58ac241	patient2	patient2	f	patient2@patient.com		678123450
ad43b4ec-5612-4636-9a6c-32aaa3553ad2	patient3	patient3	h	patient3@patient.com		123456789
cb637e41-57fc-4dfa-9eb5-f01448467ac6	patient4	patient4	h	patient4@patient.com		123456788
cd76b9f2-e6ce-4875-bce8-803d91c2b76e	patient5	patient5	h	patient5@patient.com		123456888
02f03311-0769-42ce-8bea-b6d53e6abe34	patient6	patient6	h	patient6@patient.com		123458888

Figure A.5: List of athletes

Lu

« 1 »

Sportifs inscrits:

ID	Nom	Prenom	Genre	Email	Tel	Code
0f6369d9-d96b-444a-bea8-8e36db6e3c53	Dahm	Luca	h	lucadahm@hotmail.be	0479555625	KX4SRNFGG
55debbe0-b79a-49f6-bf39-56997bf41b19	Deplechin	Luca	h	luca.dep72@gmail.com		TK0ZS4JRN
e565f968-af57-4b75-9c72-9bccd550a03b	Pringier	Luka	h	luka.pringier@hotmail.com	32492775003	L40QMJ8G9
0a387c5e-90b3-43de-9625-b513deaf417d	Lesuisse	Lucas	h	lesuisselucas@gmail.com	32472785881	0PQX6NAPF
072b7f0b-f551-4bea-98da-687a2b7732d5	Abena	Jean-Luc	h	jeanlucabena09@gmail.com	0471650537	B51PYCFF4
e2ee5308-5c0a-425c-bc17-c3f2ff78ab65	Lurkin	Thomas	h	thomaslurkin@hotmail.fr	0477747905	9C736C6AH

Figure A.6: Filtered list of athletes when searching

Enquêtes Résultats Sportifs Question./Sportifs Profil Déconnexion

ID de l'étude

ID du questionnaire

Filtrage avancé

Télécharger les resultats ☐ Afficher les resultats ☐

Générer les resultats

User	IDUser	Questionnaire	IDQuestionnaire	Start date	Start time	End date	End time
Clara	153e9ae1-d7c1-4aa6-b609-cb58612bacda	Badminton [femme]	7a6ef82e-f43c-4d1d-b2ab-3f99555018ba	2023-02-08	12:35:55.000	2023-02-14	22:27:12.000
Clara	153e9ae1-d7c1-4aa6-b609-cb58612bacda	Badminton [femme]	7a6ef82e-f43c-4d1d-b2ab-3f99555018ba	2023-02-16	10:41:21.000	2023-02-16	10:42:34.000

Figure A.7: Results page

Enquêtes Résultats Sportifs Question./Sportifs Profil Déconnexion

Gestion des membres

Ajouter un nouveau membre

Nom

Prenom

Email

Telephone

Mot de Passe
Z85PZ0QG4Y48

Role

ANNULER ENVOYER

Rechercher

« 1 »

Membre(s) inscrit(s):

ID	Nom	Prenom	Email	Tel	Role
8ed44e42-8537-4c3b-80bb-e265055da34e	admin	admin	admin@uliege.be		1
e7ff522d-49b8-4e70-ba41-bdecc35b6c69	consulting	consulting	consulting@uliege.be		3
8d66dc27-a36a-4ba0-ac63-d2223758568e	moderator	moderator	moderator@uliege.be		2
3818cc84-fbc2-435b-84ab-1c0058734fe9	Paulus	Julien	julien.paulus@live.be		1
41680f3c-e3c6-4abe-9fd5-5e11920338ec	Picard	Xavier	xavier@athletin.be		1
7b2ed56d-14f5-4a28-952d-8b28c014b470	test	admin	testadmin@uliege.be	0403042023	1

Figure A.8: Members Page

Enquêtes Résultats Sportifs Question./Sportifs Profil Déconnexion

Gestion des enquetes par sportif:

1 : patient1 patient1(321ee98d-6da3-4f97-ae6b-bcdcff0be2d3)

Inscription aux enquetes:

#	Enquete(s)	Date debut/fin
1	Enquete BO	INSCRIRE
2	Etude 1	INSCRIT
3	Etude 10	INSCRIRE
4	Etude 11	INSCRIRE
5	Etude 12	INSCRIRE
6	Etude 13	INSCRIRE
7	Etude 15	INSCRIT
8	Etude 2	INSCRIRE
9	Etude 3	INSCRIRE
10	Etude 4	INSCRIRE
11	Etude 5	INSCRIRE
12	Etude 6	INSCRIRE
13	Etude 7	INSCRIRE
14	Etude 8	INSCRIRE
15	Etude 9	INSCRIRE
16	Nouvelle enquete	INSCRIRE

2 : patient2 patient2(cd6c2c2e-9953-492d-b2e1-038ba58ac241)

Figure A.9: Register an athlete to studies

Enquêtes Résultats Sportifs Question./Sportifs Profil Déconnexion

Gestion des enquetes

Créer une enquête et l'ajouter : Nom de l'enquete Description de l'enquete +

Rechercher une enquete

Enquete 1 : Enquete BO Sportifs Questionnaires

Enquete 2 : Etude 1 Sportifs Questionnaires

Enquete 3 : Etude 10 Sportifs Questionnaires

Enquete 4 : Etude 11 Sportifs Questionnaires

Enquete 5 : Etude 12 Sportifs Questionnaires

Enquete 6 : Etude 13 Sportifs Questionnaires

Enquete 7 : Etude 15 Sportifs Questionnaires

Enquete 8 : Etude 2 Sportifs Questionnaires

Figure A.10: Study Page

Enquetes Résultats Sportifs Question./Sportifs Profil Déconnexion

Gestion des sportifs

Rechercher

« 1 2 »

Sportifs inscrits:

ID	Nom	Prenom	Inscription
321ee98d-6da3-4f97-ae6b-bcdcff0be2d3	patient1	patient1	INSCRIT
cd6c2c2e-9953-492d-b2e1-038ba58ac241	patient2	patient2	INSCRIT
ad43b4ec-5612-4636-9a6c-32aaa3553ad2	patient3	patient3	INSCRIRE
cb637e41-57fc-4dfa-9eb5-f01448467ac6	patient4	patient4	INSCRIRE
cd76b9f2-e6ce-4875-bce8-803d91c2b76e	patient5	patient5	INSCRIRE
02f03311-0769-42ce-8bea-b6d53e6abe34	patient6	patient6	INSCRIRE
f14d0b93-d2f4-4516-9e75-cb273558b0df	patient7	patient7	INSCRIRE
62ce4229-4b5d-463c-b17a-32f1a51e3d0a	patient8	patient8	INSCRIRE
cbb17da7-9414-412f-a115-d3c33113633e	patient9	patient9	INSCRIT
0cb46e18-6927-4d13-8b37-00fb6d3aca62	patient10	patient10	INSCRIRE
14025866-697f-487c-8018-5d7919e576d2	patient11	patient11	INSCRIRE

Figure A.11: Register athletes to a study

Enquetes Résultats Sportifs Question./Sportifs Profil Déconnexion

Gestion des questionnaires

Créer un questionnaire et l'ajouter : Nom du questionnaire Description du questionnaire +

Ajouter des questionnaires existants

Formulaire 1 : Questionnaire post entrainement

Description : Liste de questions posées après une séance d'entrainement

Cloner des questionnaires | Ajouter des questions | Modifier le questionnaire | Évènements

Alimentation

Description : Qualité de ton alimentation ces 24 dernières heures

Sommeil

Charge de travail scolaire/professionnel

Densité

Formulaire 2 : Badminton: Para-médical [femme]

Formulaire 3 : Badminton: Journée sans session [homme]

Formulaire 4 : Badminton: Match (double) [femme]

Figure A.12: Study Questionnaires Page

Ajouter des questions

Rechercher

« 1 2 »

Questions ajoutés:

ID	Nom	Description	Inscription
0a87cf4d-9cb3-4ddc-9e36-4a46130e7e4b	Exigences énergétiques	Sollicitations énergétiques de la session	AJOUTER
8abe7368-aaae-4ac2-b6b6-98c7a5b7da3f	Qualité technique	Maîtrise de mes gestes & de mes mouvements	AJOUTER
a69930f8-d783-4c87-abde-a4cc9c9c106f	Fatigue	Sensation de diminution des capacités suite à la session	AJOUTER
3c7d73d7-7e98-4591-b116-d1fdd50a1767	Densité	Densité des efforts au cours de la séance	Ajouté
2182b194-4125-427b-af7f-8bde89cbccb8	Nationalité de l'adversaire		AJOUTER
d51c59af-8ca6-430e-b6b9-ebc1c3263bb5	Charge de travail scolaire/professionnel	Quantité de travail à accomplir	AJOUTER
812b6a85-3ce1-45cb-a9a2-a906e5877e29	Charge de travail scolaire/professionnel	Quantité de travail à accomplir	Ajouté
03ae4aea-2f66-4e80-9a1b-4f58eda1ac06	Nervosité	Etat d'irritation & d'instabilité nerveuse	AJOUTER
352198a4-0aea-451e-9602-c9f98085bbf2	Intensité moyenne	Moyenne de l'intensité des efforts	AJOUTER
9cc8684f-6cf6-440b-8715-f8d7eb26164a	Contexte de pratique		AJOUTER
85cdd15f-710c-430b-a7df-ebb8fd64c131	Qualité technique	Maîtrise de mes gestes & de mes mouvements	AJOUTER
47025a9f-803d-4b16-9ff7-ab50b3a359f1	Nom de l'adversaire		AJOUTER
2dc9d251-5831-497d-a684-26c132ec171a	Blessure	Situation lésionnelle	AJOUTER
b15578a0-d5ca-45bf-b741-627bb3492b40	Animation pédagogique	Capacité de l'entraineur à adapter les situations à l'objectif	AJOUTER
e2b4433c-c7ec-46c6-a56f-96834aa147e3	Blessure	Mode d'apparition	AJOUTER
ba0f47dd-559f-4907-8759-8732fe989cbc	Exigences psychologiques	Sollicitations cognitives & émotionnelles de la session	AJOUTER

Figure A.13: Add Question Page

Ajouter des questionnaires

Rechercher

« 1 2 »

Questionnaires ajoutés:

ID	Nom	Description	Cloner
a3be49e4-8bf0-46e1-a26b-f30e676bfe93	Badminton: Blessure - Douleur		Cloner
1cd9e518-3f4e-405b-8b34-009041be5fee	Badminton: Para-médical [femme]	Séance de soins	Cloner
e007dccc-d9a2-40d6-b7f3-5033a9742bbc	Badminton: Journée sans session [homme]		Cloner
8d0f88c7-eb49-4c21-950c-b6d80eebeb17	Rugby: Autre [femme]		Cloner
31206a1a-b2ad-47d9-9009-14c9758ab11b	Badminton: Match (double) [femme]		Cloner
009b7682-4ac7-4572-a995-4cccb40d38f3	Badminton: Talk [homme]		Cloner
321cee5e-cc2a-452b-b0c3-105c9296972a	Rugby: Autre [homme]		Cloner
87755a22-9577-4445-ad7e-c6981ca02feb	Badminton: Journée sans session [femme]		Cloner
1f492b34-1070-4886-9f1d-0b00c4443995	Rugby: Journée sans session [femme]		Cloner
3f0c852d-3786-4982-8af5-3771d2b26c65	Rugby [homme]	Entrainement technico-tactique	Cloner
b50ae120-ff81-46f3-b8d7-59a3ab6c7479	Badminton: Match (simple) [femme]		Cloner
eccbdd89-e651-4aea-b5f6-d2434778f9cd	Badminton: Blessure - Douleur [suivi]		Cloner
9ab2b532-bfee-4cf3-8d51-739a3309bd38	Badminton: Talk [femme]		Cloner
bc9f6492-86d6-4afa-98d0-7a57fd0ba240	Rugby: Para-médical [femme]	Séance de soins	Cloner
1788c8a5-7a6a-4b83-88c5-f5fb00180eda	Rugby: Blessure - Douleur [suivi]		Cloner
5ef148fd-7396-42b4-9232-3b47c12ce504	Rugby: Match [femme]		Cloner
e9fb6bf2-77bf-47ec-83cc-09cf0824f995	Rugby [femme]	Entrainement technico-tactique	Cloner

Figure A.14: Add Questionnage Page

Enquetes Résultats Sportifs Question./Sportifs Profil Déconnexion

Gestion des évènements

Heure fixe | Aléatoire | Quotidien

Évènement à heure fixe

Date: 08/06/2023

Heure de début : 11:37 Heure de fin : 12:37 + Ajouter la plage horaire

Créer l'évènement

Figure A.15: Daily Fixed Event Page

Figure A.16: Daily Random Event Page

Enquetes Résultats Sportifs Question./Sportifs Profil Déconnexion

Gestion des évènements

Heure fixe | Aléatoire | Quotidien

Évènement quotidien

Delta 0 | Delta F | 08/06/2023 Modifier la date | Aucun

Lun Mar Mer Jeu Ven Sam Dim Tous les jours

Ajouter un évènement fixe | Ajouter un évènement aléatoire

Evènement(s)

Evènement 1 | Period 1 | 11:37 - 12:37

Créer l'événement

Figure A.17: Daily Event Page

Bibliography

[1] *ATHLETin, agile athlete thinking management.* https://athletin.io/index.php/web-application/.

[2] Lodrini Guillaume. *Master thesis : ATHLETin: Web module for the management of athletes' training calendar and medical appointments.*

[3] *PostgreSQL Documentation.* https://www.postgresql.org/docs/current/.

[4] *Go Programming Language.* https://go.dev/.

[5] Rob Pike. *Another Go at Language Design.* https://web.stanford.edu/class/ee380/Abstracts/100428-pike-stanford.pdf.

[6] Jinzhu Zhang. *GORM Guides.* https://gorm.io/.

[7] Lokesh Gupta. *What is REST.* https://restfulapi.net/.

[8] Google. *Dart programming language.* https://dart.dev/.

[9] Sneath Tim and Walrath Kathy et al. *Dart Overview.* https://dart.dev/overview.

[10] Google. *The Dart type system.* https://dart.dev/language/type-system.

[11] Sheldon Robert. *What is garbage collection (GC)?* https://www.techtarget.com/searchstorage/definition/garbage-collection.

[12] Sullivan Matt. *Flutter: Don't Fear the Garbage Collector.* https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30.

[13] Google. *Dart Null Safety.* https://dart.dev/null-safety.

[14] Bizzotto Andrea. *Dart Null Safety: The Ultimate Guide to Non-Nullable Types*. https://codewithandrea.com/videos/dart-null-safety-ultimate-guide-non-nullable-types/.

[15] Google. *dartdevc: The Dart development compiler*. https://dart.dev/tools/dartdevc.

[16] Google. *Asynchronous programming: futures, async, await*. https://dart.dev/codelabs/async-await.

[17] Google. *Core libraries*. https://dart.dev/guides/libraries.

[18] *Flutter framework*. https://flutter.dev/.

[19] Google. *Flutter architecture overview*. https://docs.flutter.dev/resources/architectural-overview.

[20] Google. *Introduction to widgets*. https://docs.flutter.dev/ui/widgets-intro.

[21] Google. *StatelessWidget class*. https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html.

[22] Google. *StatefulWidget class*. https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html.

[23] Abhishek Doshi. *Widget lifecycle — Flutter!* https://abhishekdoshi26.medium.com/widget-lifecycle-flutter-3db5d824d033.

[24] Google. *createState abstract method*. https://api.flutter.dev/flutter/widgets/StatefulWidget/createState.html.

[25] Google. *initState method*. https://api.flutter.dev/flutter/widgets/StatefulWidget/createState.html.

[26] Google. *build abstract method*. https://api.flutter.dev/flutter/widgets/State/build.html.

[27] Google. *setState method*. https://api.flutter.dev/flutter/widgets/State/setState.html.

[28] Google. *didUpdateWidget method*. https://api.flutter.dev/flutter/widgets/State/didUpdateWidget.html.

[29] Google. *Flutter dispose method*. https://api.flutter.dev/flutter/widgets/State/initState.html.

[30] Pankaj Tyagi. *Lifecycle of Stateful Widget* . https://mobikul.com/lifecycle-of-stateful-widget/.

[31] Google. *InheritedWidget class*. https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html.

[32] Google. *Provider*. https://pub.dev/packages/provider.

[33] Google. *ChangeNotifier class*. https://api.flutter.dev/flutter/foundation/ChangeNotifier-class.html.

[34] Google. *notifyListeners method*. https://api.flutter.dev/flutter/foundation/ChangeNotifier/notifyListeners.html.

[35] Ayush Pawar. *Exploring State Management in Flutter with Provider*. https://cswithiyush.hashnode.dev/exploring-state-management-in-flutter-with-provider.

[36] Google. *MultiProvider class*. https://pub.dev/documentation/provider/latest/provider/MultiProvider-class.html.

[37] Google. *Hot reload*. https://docs.flutter.dev/development/tools/hot-reload.

[38] Vyacheslav Egorov. *Introduction to Dart VM*. https://mrale.ph/dartvm/.

[39] encrypter09. *Advantages and Disadvantages of Flutter's Hot Reload Feature*. https://www.geeksforgeeks.org/advantages-and-disadvantages-of-flutters-hot-reload-feature/.

[40] Google. *Flutter Navigation*. https://docs.flutter.dev/ui/navigation.

[41] Prof. L. Mathy, G. Gain, and V. Rossetto. *Development of a web module for ATHLETin: Technical description*.

[42] *Docker*. https://docs.docker.com/get-started/.

[43] Silverlock Matt. *Gorilla mux package*. https://github.com/gorilla/mux.

[44] *Swaggo*. https://github.com/swaggo/swag.

[45] *Introduction to JSON Web Tokens*. https://jwt.io/introduction.

[46] *Postman*. https://www.postman.com/.

[47] Glenn E. Krasner and Stephen T. Pope. *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*. https://web.archive.org/web/20100921030808/http://www.itu.dk/courses/VOP/E2005/VOP2005E/8_mvc_krasner_and_pope.pdf.

[48] Josh Smith. *Patterns - WPF Apps With The Model-View-ViewModel Design Pattern*. https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern.

[49] Zimmermann Max. *What is the difference between MVC and MVVM (for Flutter)?* https://medium.com/@m-zimmermann1/what-is-the-difference-between-mvc-and-mvvm-for-flutter-136cb33afc1.

[50] Google. *DevTools*. https://docs.flutter.dev/tools/devtools/inspector.

[51] Google. *Using the Flutter inspector*. https://docs.flutter.dev/tools/devtools/overview.

[52] Google. *Using the debugger*. https://docs.flutter.dev/tools/devtools/debugger.

[53] Google. *Chrome DevTools Overview*. https://developer.chrome.com/docs/devtools/overview.

[54] Google. *Lighthouse: Optimize website speed*. https://developer.chrome.com/docs/devtools/lighthouse/.

[55] Tushar Pol. *Google Lighthouse: What It Is How to Use It*. https://www.semrush.com/blog/google-lighthouse/.

[56] Google. *Testing Flutter apps*. https://docs.flutter.dev/testing.

[57] Mohit Joshi. *Testing in Flutter*. https://medium.flutterdevs.com/testing-in-flutter-fd0f82ecddc7.

[58] *Debian*. https://www.debian.org/index.en.html.

[59] *Nginx*. https://www.nginx.com/.